AFRL-IF-RS-TR-2003-183
Final Technical Report
August 2003

ROBUST LOCALLY OPTIMUM DETECTION IN AUTO-REGRESSIVE NOISE

Illinois Institute of Technology

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** AUGUST 2003	**3. REPORT TYPE AND DATES COVERED** Final Jan 98 – Jan 01

4. TITLE AND SUBTITLE
ROBUST LOCALLY OPTIMUM DETECTION IN AUTO-REGRESSIVE NOISE

5. FUNDING NUMBERS
C - F30602-98-C-0032
PE - G9G136
PR - 4519
TA - 42
WU - 03

6. AUTHOR(S)
Michael R. Banys and Donald R. Ucci

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Illinois Institute of Technology
Department of Electrical and Computer Engineering
Chicago Illinois 60616-3793

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/IFGC
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2003-183

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: John J. Patti/IFGC/(315) 330-3615/ John.Patti@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

This report represents the culmination of a study focusing on more extensive characterizations of a robust autoregressive locally optimum (ARLO) detector for spread spectrum signals operating in a variety of noise environments. The signal is modeled as corrupted by both Gaussian and Non-Gaussian noise as representative of many interference environments. The normal linear correlator is sub-optimum, as is a fixed non-linearity. The ARLO algorithm, in contrast, uses an adaptive nonlinearity that utilizes probability density function (pdf) estimation techniques to alleviate the need for a priori knowledge of the channel noise characteristics. The ARLO algorithm uses pdf estimation techniques for Independent Identically Distributed (IID) noise samples as the input sequence to the AR model is IID. The goal and results of this effort described herein is to further characterize performance improvements using a more efficient C (rather than MatLab) program.

14. SUBJECT TERMS
Spread Spectrum, Noise, Gaussian, Non-Gaussian, Autoregressive Locally Optimum, ARLO, Independent Identically Distributed, IID

15. NUMBER OF PAGES
77

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

List of Figures

Executive Summary

Reliable transmission of information is an important issue in military and commercial communication systems. Even though modern techniques provide some error performance improvement, systems experiencing large interference can undergo unacceptable degradation. The principal investigator and his research team, under previous contracts with the United States Air Force through Rome Laboratory, developed a system that significantly improves performance in severely degraded channels. This report represents the culmination of a study focusing on more extensive characterizations of a robust *autoregressive locally optimum* (ARLO) detector operating in a variety of noise environments.

In a communication system, undesired signals (such as channel noise, jammers, or cross-channel interference) are a primary cause of signal corruption and performance degradation. If the channel were corrupted with only additive white Gaussian noise, the chosen detector would be a linear correlator, as it is an optimum detector for this scenario. However, a linear correlator is usually sub-optimum for other types of noise, such as would be encountered in an intentionally or unintentionally hostile environment or in a multi-user application. In this scenario, traditional techniques usually precede the linear correlator with a fixed nonlinearity [1], whose purpose is to negate the effects of the additional noise, and thus improve the performance of the linear correlator. The ARLO algorithm, in contrast, uses an adaptive nonlinearity that utilizes *probability density function* (pdf) estimation techniques to alleviate the need for *a priori* knowledge of the channel noise characteristics. The mathematical basis for the algorithm is as follows.

Consider a system that has a received signal vector, $\mathbf{r}$, given by the following expression,

$$\mathbf{r} = \mathbf{s}_m + \mathbf{n}, \tag{1}$$

where $\mathbf{s}_m$ is the transmitted signal, $m = 0 \,\text{or}\, 1$, and $\mathbf{n}$ is the total noise signal (interference plus background noise).

Letting the observed value of $\mathbf{r}$ be ρ, the LO nonlinearity used in the ARLO detector is,

$$g_i(\rho) = -\frac{\frac{\partial}{\partial \rho_i} f_{\mathbf{n}}(\rho)}{f_{\mathbf{n}}(\rho)}, \tag{2}$$

where $f_{\mathbf{n}}(\cdot)$ is the joint pdf of the noise.

Therefore, the LO detector likelihood function, $l\left(\rho\right)$, for the detection of known binary signals can be written as,

$$l\left(\rho\right) = \textstyle\sum_{i=1}^{N}\left[\left(s_{1i} - s_{0i}\right) g_i\left(\rho\right)\right] \quad \begin{array}{c}\text{choose } H_1\\ > \\ < \\ \text{choose } H_0\end{array} \quad \gamma \tag{3}$$

where γ is an appropriately chosen decision threshold.

Chapter 1

Introduction

In modern military and commercial communication systems, it is imperative that information be transferred in a reliable and secure manner. Current techniques, such as those used in *spread spectrum* (SS) systems, provide a measure of security and reliability. Systems that undergo extreme interference, however, can exhibit unacceptable performance. It is, therefore, imperative to investigate techniques to combat the degradation experienced by such systems. Since the decrease in acceptability is caused by interference, it is logical to develop methods to mitigate these undesirable signals.

Locally optimum (LO) detection is a technique that lends itself to such situations. Its performance is close to that of the optimum Bayesian or *globally optimum* (GO) detector with a much reduced complexity. It is also robust in that it can be implemented to adapt to changing environments. Previous work [1] has focused on such LO detectors that utilize estimates of the *probability density function* (pdf) alleviating the need for any a priori knowledge of the noise characteristics. Multidimensional signal processing can improve performance in these systems. However, the complexity increases enormously as the dimensionality of the *joint pdf* (jpdf) of the signaling environment grows. More recently, techniques have centered on the use of *autoregressive* (AR) modeling methods to better estimate the noise environment and enhance performance. The *Autoregressive Locally Optimum* (ARLO) detector is one that successfully combines these techniques. The use of AR process models greatly reduces the dimensionality, and hence the complexity, of the resulting ARLO detector. The statistics of the jpdf remain an issue. To this end, pdf estimation techniques for *independent identically distributed* (iid) noise samples prove useful since the input sequence to the AR model is iid albeit characterized by some unknown pdf. The necessary AR model parameters are determined by well-known spectral estimation techniques. Thus, the goal of this research effort is to further characterize performance improvements in robust LO detectors using AR techniques.

In pursuit of this goal, simulations were initially performed using MATLAB® and later re-coded into *C*. The simulation uses B total received symbols sampled at a rate of N_b samples/symbol. Standard AR model estimation techniques are applied to all of the received data. The *bit error performance* (BER) of five detectors are computed and compared. The detectors studied are the linear correlator, the LO detector without memory, the linear correlator and the LO detector operating on whitened noise, and the ARLO detector. A block diagram illustrating this system is shown in Figure 1.1.

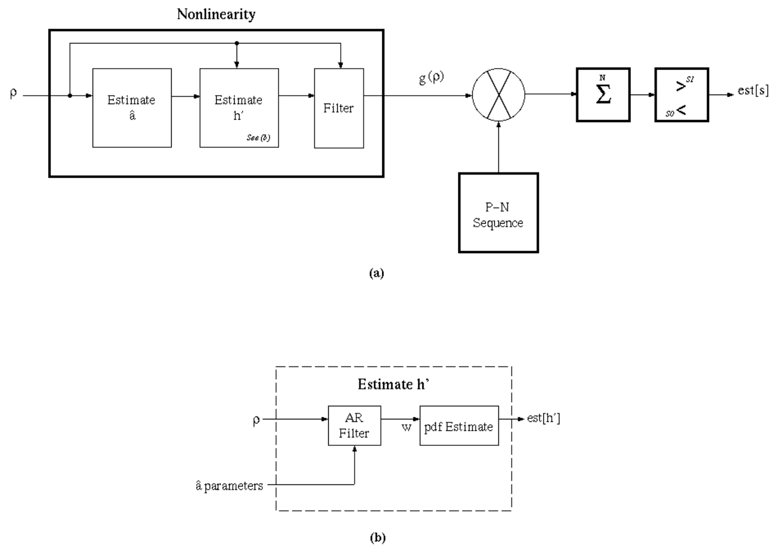

Figure 1.1: (a) ARLO system block diagram. (b) Expanded view of *Estimate h'* block.

The interference environment was generated using an AR process by passing iid noise through a given AR model (i.e., the $\{a_i\}$ are known). The communication system employed *binary phase shift keyed* (BPSK) signals so that each symbol represented a bit. The AR process model used a 16th order filter. The spread spectrum system used a processing gain of 16 chips/bit and the number of samples/chip, N_c, was set to 4. The noise pdf was estimated using the histogram method. A Monte Carlo simulation was performed using a number of iterations.

The organization of this report is as follows. First, pdf estimation techniques are evaluated for performance enhancement in Chapter 2. Next, the performance of the ARLO detector is compared with other standard detectors in multiple interference environments in Chapter 3. Additionally, this chapter reviews the effect of detector variables on performance. Chapter 4 focuses on the conversion of the simulation into a high level language. In addition, this chapter analyzes the execution speed of the simulation and reviews the effect of detector parameters on said speed. Finally, a summary of the research study is provided in Chapter 5, including a discussion concerning future research and development of the ARLO detector.

Chapter 2

Probability Density Function Estimation

The current form of the simulation uses the histogram method for estimating the pdf of the whitened received signal. The derivative of the resulting pdf is then used to generate the decision function, $g(\rho)$, for choosing between the two BPSK symbols. Simulation results have revealed that a large portion of the processing time is devoted to the calculation of $g(\rho)$. Therefore, the goal was to determine if a faster, more efficient alternative to the histogram method could be found for estimating the pdf of the received signal.

Efforts became focused on estimating the unknown pdf of a correlated data sequence via POCS methods. However, after initial research, it was determined that the POCS method may not offer the coding simplicity and speed required due to its potential complexity. Research was then concentrated on using *artificial neural networks* (ANN) for pdf estimation [5]. This work proposes that a mixture of *radial basis functions* (RBF) can approximate an unknown pdf. An RBF ANN can be trained in such a way that it outputs the approximated pdf, even when the input data is correlated. However, in this form, the ANN method was more complicated than necessary due to the fact the actual simulation data is uncorrelated because the received signal is whitened prior to the determination of its pdf. This fact simplifies the problem at hand, thus making it possible to use a similar but less complicated theory called Parzen's Estimator [4].

2.1 Histogram Method

The histogram method attempts to estimate the pdf of a signal through the use of a histogram to approximate its continuous pdf. Each signal sample is deposited into the appropriate bin, based on its value. This process creates a histogram which describes the pdf of the signal. The greater the number of bins, the better the histogram represents the continuous pdf of the actual received signal. This method is very simple to implement, and quite accurate relative to its level of complexity.

2.2 Parzen's Estimator

Parzen's estimator attempts to estimate a pdf through the use of *radial basis functions* (RBF). RBFs are parametric functions which satisfy the axiomatic properties of pdfs. They assume a maximum value at zero radius and monotonically decay from this maximum as the radius increases [5]. For instance, the Gaussian density function is considered to be an RBF. Parzen's estimator proposes that the estimated pdf of a set of correlated data is the weighted sum of individual RBFs centered at each data point. This can be written as,

$$\hat{f}_X = f_{X_1,\ldots,X_n}\left(x|x_1,\ldots,x_n\right) \overset{\triangle}{=} \frac{1}{nh\left(n\right)} \sum_{i=1}^{n} k\left(\frac{x-x_i}{h\left(n\right)}\right), \tag{2.1}$$

where n is the number of data points, $k\left(\cdot\right)$ is the RBF, and $h\left(n\right)$ is a smoothing factor [4]. Currently, the Gaussian function is being used as the RBF, $k\left(\cdot\right)$,

$$k\left(y\right) = \frac{1}{\sqrt{2\pi}} e^{\frac{-y^2}{2}}, \tag{2.2}$$

and $h\left(n\right)$ is the inverse p^{th} root of the sample size [4],

$$h\left(n\right) = n^{\frac{-1}{p}}. \tag{2.3}$$

The above definition of the smoothing factor results in the fact that as the value of p increases, the bias of the estimator increases, while the variance of the estimator decreases. If p takes on a value of 2, the resulting estimator is unbiased. As stated above, the derivative of the estimated pdf must be taken in order to determine the $g\left(\rho\right)$ function, so achieving a smooth pdf is important. However, the resulting unbiased estimator, when $p = 2$, is not smooth. Smoothness is achieved when the value of p is greater than 3, but these estimators are biased. Comparatively, in the histogram method, the number of bins used affects the smoothness of the resulting pdf. Currently, the simulation employs a histogram with thirty-three bins. This results in a jagged pdf. Smoother results are achieved when the number of bins approaches twelve. The various values of p yield approximately the same estimate of the pdf. When $p = 3.4$, the lowest MSE is achieved. However it is only slightly smaller than the other MSE values. All of the histogram estimates are somewhat jagged at transition points, thereby resulting in large MSEs.

The Parzen estimator can prove to be a reliable estimator of the pdf of the whitened received signal data. The method results in smooth, estimated pdfs with low MSEs. In order to make a definitive decision on whether the Parzen estimator is more effective than the histogram method, further research needs to be conducted. The issues which need to be addressed are the following: the exact estimation time of each method, the capabilities of each method to reliably estimate pdfs using fewer data points, and the capacity of each method to estimate other types of non-Gaussian pdfs.

Chapter 3

Performance Analysis

The ultimate measure of the total system performance of a communication system is the BER, the ratio of erroneous bits received to the total number of bits transmitted. In order to be able to measure the BER, a SS communication system was previously implemented using MATLAB® as a simulation tool [3]. The simulation will allow us to study the BER performance of the system in varying interference environments as well as under different configurations of system parameters. The first section of this chapter will describe the various interference environments implemented in the simulation. The second section will analyze the detector performance against each interferer. The ARLO detector has a number of parameters whose values can affect the performance of the communication system. Therefore, the third section of this chapter will discuss in detail the effects of choosing alternate parameter values.

3.1 Interference Environments

Four different interferers were implemented for use in the simulation, *continuous wave* (CW), *partial band* (PB), *dual continuous wave* (2CW), and *mixed* interferers.

3.1.1 Continuous Wave

The CW interferer is simply a sine wave of a predetermined frequency, with zero phase shift, as shown in Figure 3.1. The amplitude of the sine wave is determined by the *interference to signal ratio* (ISR) as,

$$Amplitude = \sqrt{\frac{2\left(10^{\frac{ISR}{10}}\right)}{I}}, \tag{3.1}$$

where I is the number of CW interferers (one, in the single CW case).

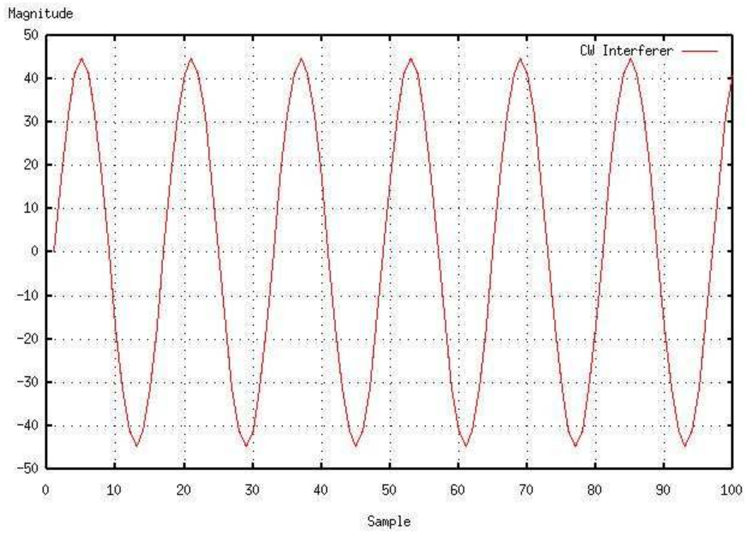

Figure 3.1: CW Interferer, 100 data samples.

The frequency of the sine wave is determined as,

$$f = \mathrm{N_c B}, \tag{3.2}$$

where $\mathrm{N_c}$ is the number of samples per chip and B is the number of symbols transmitted over the channel. The power spectra of a CW interferer with ISR = 30 dBW is shown in Figure 3.2.

3.1.2 Partial Band

The PB interferer exhibits a large degree of power over a chosen band of frequencies. This jammer is created by passing an *independent, identically distributed* (IID) white noise process through a predetermined AR filter, which is created by taking the polynomial,

$$poly = [1 \;\; -0.75]\,, \tag{3.3}$$

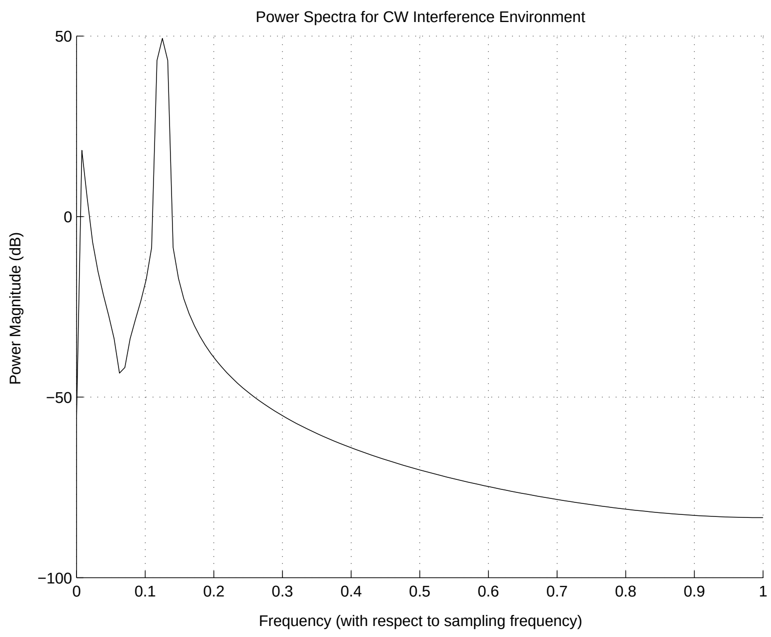

Figure 3.2: CW power spectra.

and convolving it with itself four times. This "auto-convolution" process creates a 16th-order filter with filter coefficients,

$$a_t = \begin{bmatrix} -1 \\ 12 \\ -67.5 \\ 236.25 \\ -575.859 \\ 1036.55 \\ -1425.25 \\ 1527.06 \\ -1288.45 \\ 858.969 \\ -450.959 \\ 184.483 \\ -57.651 \\ 13.3041 \\ -2.13815 \\ 0.213815 \\ -0.0100226 \end{bmatrix}. \tag{3.4}$$

The power spectra of the filter described by (3.4), with ISR $= 30\,\mathrm{dBW}$, is shown in Figure 3.3.

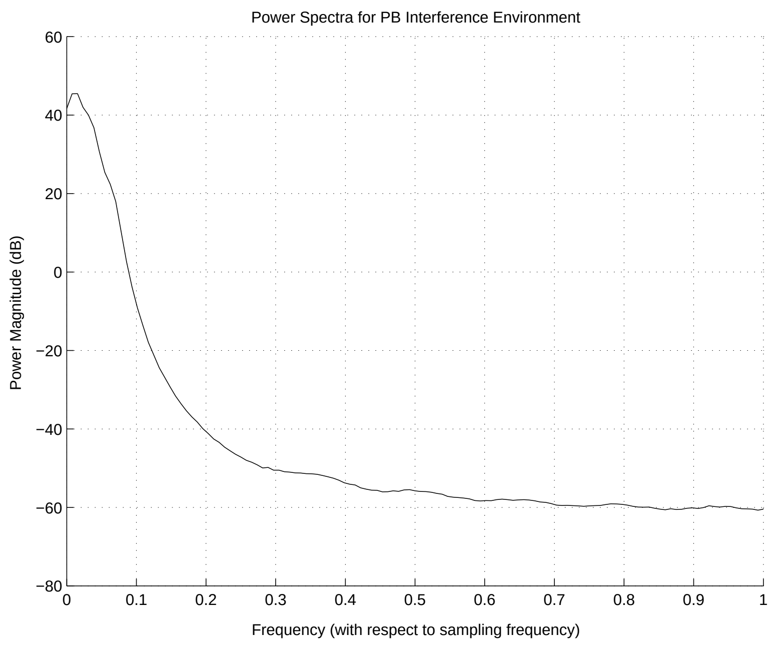

Figure 3.3: PB power spectra.

3.1.3 Dual Continuous Wave

The 2CW interferer is easily created by summing two single CW interferers. The amplitude of each sine wave is determined as shown in (3.1), with I = 2. The frequency of each sine wave is calculated by,

$$f = \begin{bmatrix} \mathrm{N_cB} \\ 0.3\mathrm{N_cB} \end{bmatrix}, \tag{3.5}$$

with the variables the same as in (3.2). Figure 3.4 shows the 2CW interferer, with ISR = 30 dBW, along with its two component sine waves. The power spectra for the 2CW interferer is shown in Figure 3.5.

3.1.4 Mixed

The mixed interferer is created by summing a CW interferer with a PB interferer. The ISR of the combination is fixed; the sum of the ISR's of the components equals the value chosen. The power spectra of such a jammer is shown in Figure 3.6.

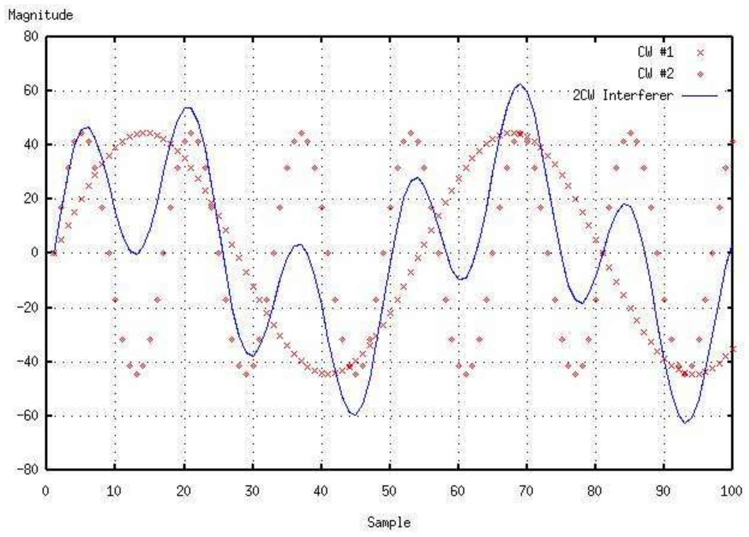

Figure 3.4: 2CW interferer, shown with its component parts.

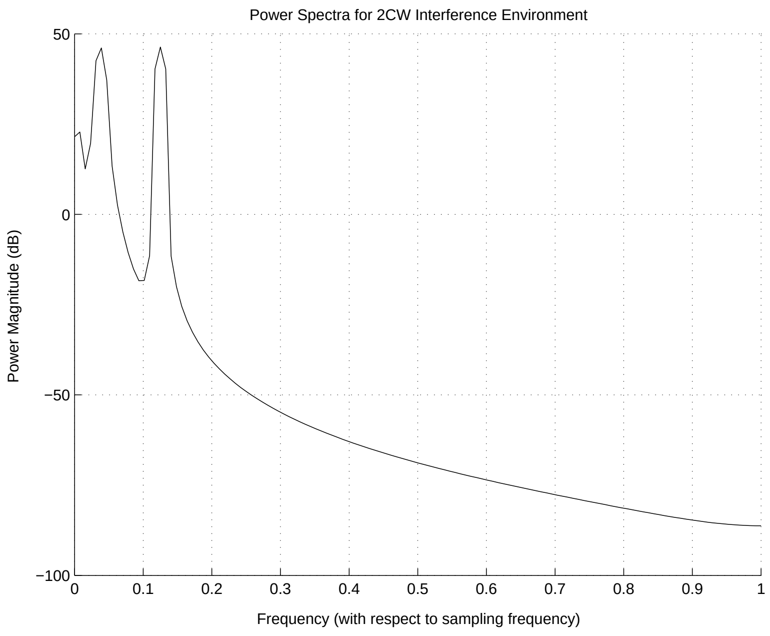

Figure 3.5: 2CW power spectra.

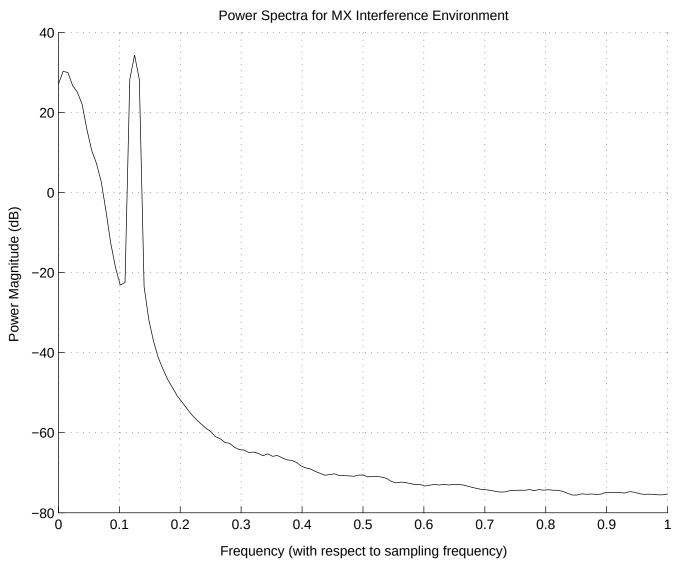

Figure 3.6: MX power spectra.

3.2 Detector Performance

The SS communication system simulation has a number of parameters that define the system operation. In order to properly analyze the performance of the ARLO detector, and compare it to the BER performance of other detectors, default values for each parameter must be chosen. Assessing the effect on performance of changing these parameters is the focus of the third section of this chapter. Default values were chosen based on various factors such as execution time and typical values encountered in practice. Table 3.1 shows the default variables chosen for simulation purposes. All simulations can be assumed to have been executed with these parameter values, unless otherwise noted.

Four other detectors are implemented in the simulation for comparison with the ARLO detector. The simplest is the *linear correlator* (LC), which is the optimum detector in an additive white Gaussian noise environment, such as the thermal background noise used in the simulation. The next detector is the LC on Whitened Noise (LCW). This detector performs as a linear correlator, but it operates on the received signal data stream after it has been whitened to reduce the noise component. The third detector is the LO detector, which is the basis for the ARLO detector. The final detector is the LO on Whitened Noise (LOW). This detector performs as a LO detector, but it operates on the received signal data stream after it has been whitened to reduce the noise component. The following sections will analyze the performance of the ARLO detector within each interference environment, including a scenario with no interferer present, using the default system parameters as shown in Table 3.1.

Type	Parameter	Parameter Description	Value
pdf Estimation	Bh	Total symbols to calculate histogram	128
	K	Data points to calculate	33
	T	Range of support for pdf	1000
Spread-Spectrum	Nc	Samples per chip	4
	PG	Processing Gain (chips per bit)	16
	Nb	Samples per symbol	$(PG * Nc)$
	B	Symbols transmitted/received	16,384
	Ntot	Total samples in data stream	$(B * Nb)$
AR-model	Bar	Total symbols to estimate AR parameters	$\frac{Bh}{64} = 2$
	Bg	Total symbols to calculate $\mathbf{g}(.)$	Bh
	P	Number of AR coefficients	16
Parameters	sigmat2	Variance of thermal noise	1
	limit	Limiter on noise	hard
Interferer	ISR	Interferer-to-signal ratio	30 dB

Table 3.1: Default parameters for communication system simulations. *All simulations can be assumed to have been executed with these parameter values, unless otherwise noted.*

3.2.1 Partial Band Interferer

The PB jammer provides for an extremely hostile environment for a communication system channel, as shown in the power spectra plot in Figure 3.8. The interferer overpowers the signal by nearly 40 dBW at peak jammer power, and further overloads the signal over a significant portion of the signal's main lobe. As can be seen from Figure 3.7, the ARLO detector greatly outperforms the other detectors, even the LO detector upon which it is based. At a *signal-to-thermal-noise ratio* (SNR) of 0 dB, the ARLO detector is able to perform at a BER of less than 10^{-2}, or less than 1 bit error for every 100 transmitted, despite the powerful PB jammer. The next best detector returns a BER of greater than 10^{-1}, or approximately 19 bit errors for every 100 transmitted. In this scenario, ARLO is clearly the best detector of all those tested.

3.2.2 Continuous Wave Interferer

In the CW interference environment, it can be seen from Figure 3.9 that the ARLO detector performs adequately, albeit slightly worse, than the other detectors in the less-hostile CW environment, with the exception of the LC. At 0 dB, the ARLO detector is registering a BER of $10^{-3.4}$, or 0.04 bit errors for every 100 transmitted. The other detectors' performance falls in the range of $10^{-4.5}$ to 0 BER. While ARLO is outperformed in this scenario, it still provides excellent detection capabilities.

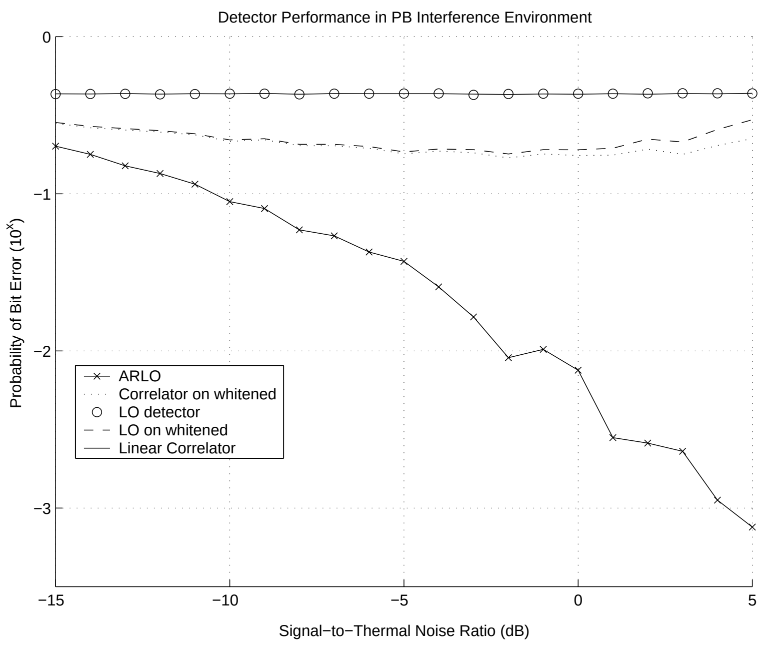

Figure 3.7: BER in PB interference environment.

3.2.3 Dual Continuous Wave Interferer

The 2CW jammer provides a greater hindrance to communication over a channel than does the CW jammer, as can be seen from the greater frequency coverage of the interferer in Figure 3.12, compared with that of the CW interferer in Figure 3.10. Therefore, one would expect that BER performance would decrease as compared to the single CW. For all five detectors, this is the case, with the four non-ARLO detectors registering a 0 dB BER in the 10^{-1} to $10^{-\frac{1}{2}}$ range, meaning $10 - 32$ bit errors for every 100 bits transmitted. The ARLO detector's 0 dB performance registered at slightly better than 10^{-2} BER, or less than 1 error out of every 100 bits. ARLO was clearly the best performer in this scenario.

3.2.4 Mixed Interferer

The MX interference scenario is potentially the most hostile environment in the simulation. While the ISR remains constant in each environment, the frequency band coverage of the MX interferer is greater than that of the other 3 jammers. ARLO again clearly outperforms the other detectors, registering a BER of $10^{-2.6}$, or 0.25 bit errors for every 100 bits transmitted, at the 0 dB SNR point. The nearest other detector experiences 10 errors for every 100 bits.

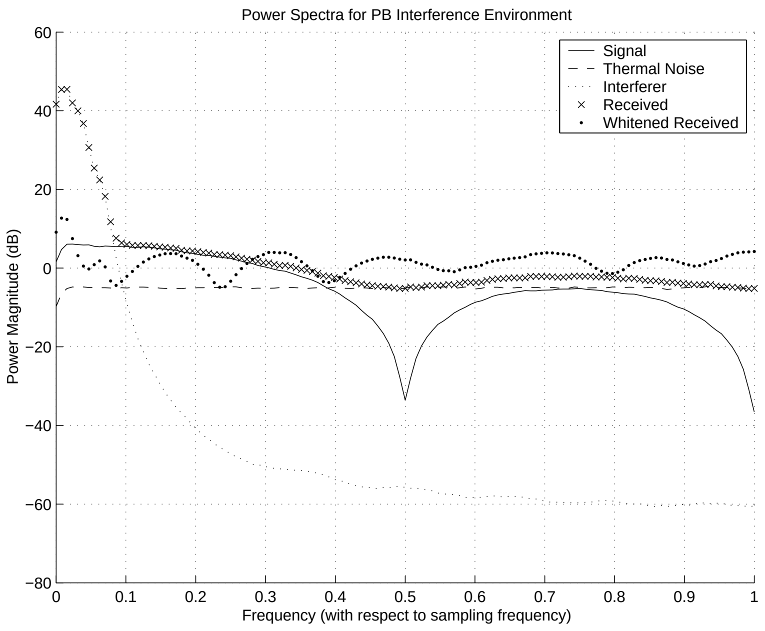

Figure 3.8: Power spectra for PB interference environment.

3.2.5 No Channel Interference

The purpose of running a simulation with no interferer present is to document the effectiveness of the ARLO detector in a very stable environment. This represents using the ARLO detector in a benign scenario. Theory predicts that the linear correlator will perform best, as it is the optimum detector in additive white Gaussian noise. The BER curve in Figure 3.15 agrees with theory, with the LC detector performing the best. ARLO performs only slightly less effectively, achieving a zero BER at all SNR greater than (-1) dB. These results indicate that the ARLO detector can be effectively used even when jamming is not present.

3.2.6 Performance Summary

Figure 3.16, a summary of section 3.2, shows that the ARLO detector clearly outperforms the other three detectors in three of the four interference environments tested. In the fourth environment, consisting of the CW interferer, the best detector achieves zero bit errors, while ARLO performs admirably with only 0.04 bit errors for every 100 bits transmitted. Additionally, the ARLO detector never exceeds 1 bit error for every 100 transmitted bits, while the best of the other detectors makes 10 bit errors 75% of the time. These results clearly indicate the performance enhancement that the ARLO detector can provide

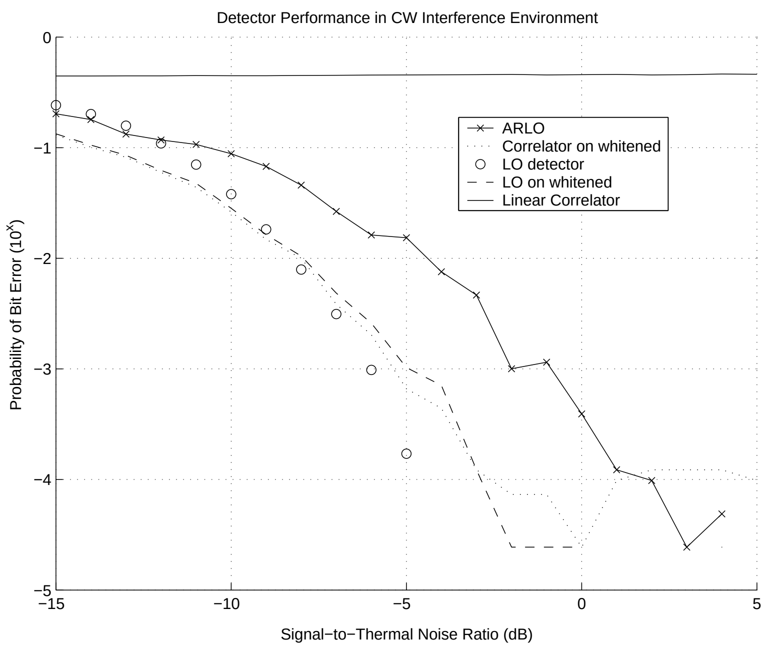

Figure 3.9: BER in CW interference environment.

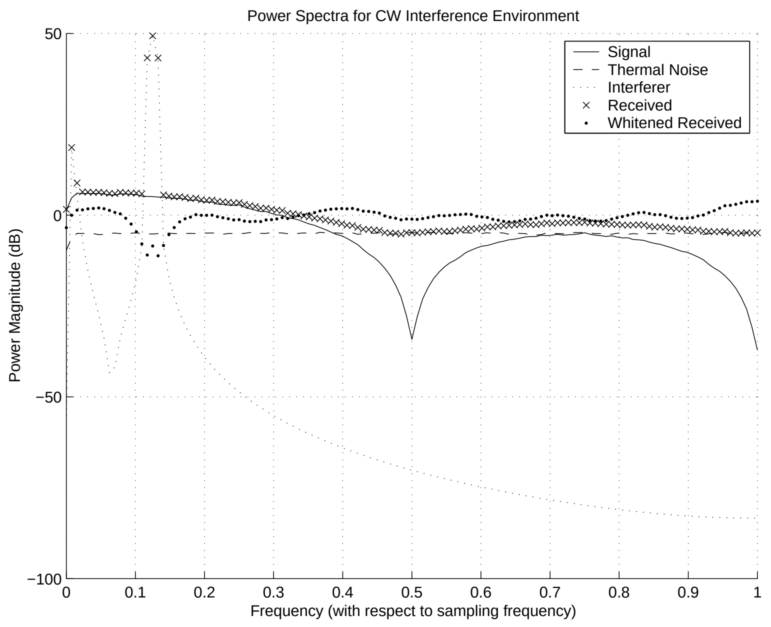

Figure 3.10: Power spectra for CW interference environment.

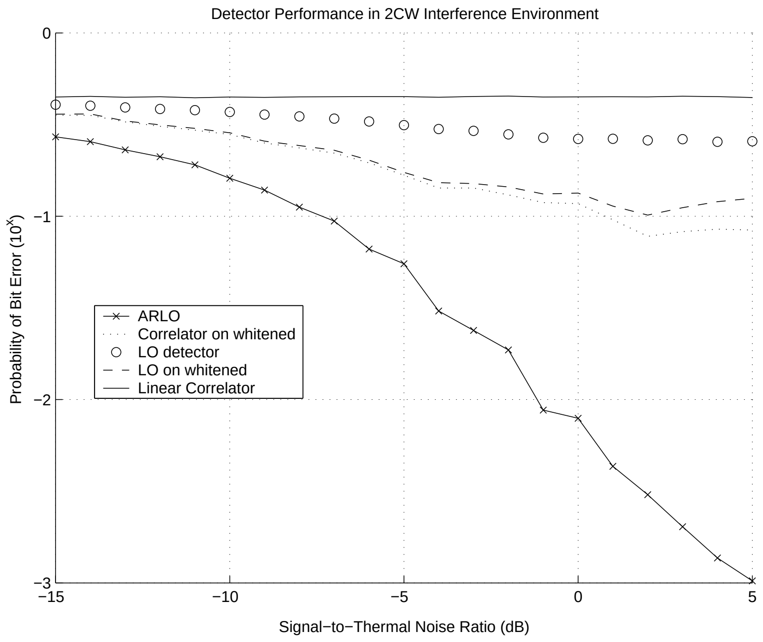

Figure 3.11: BER in 2CW interference environment.

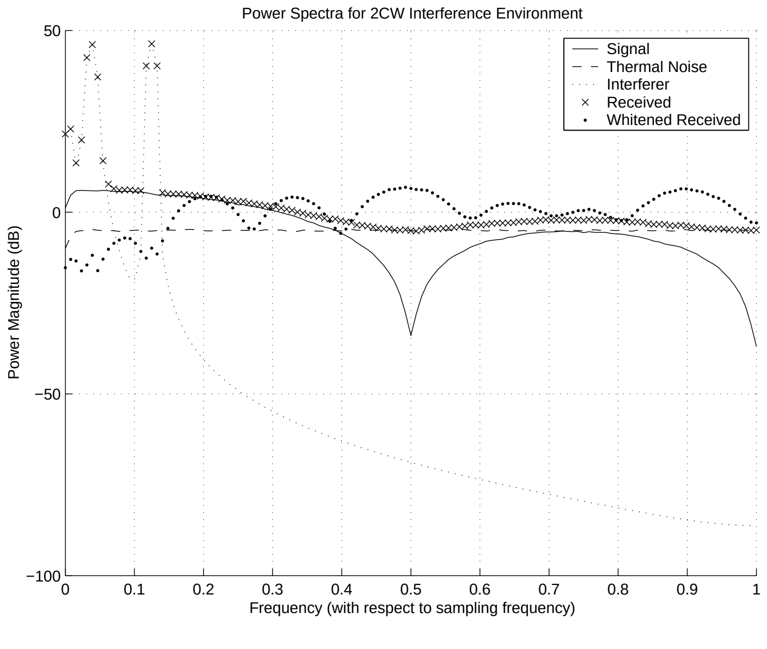

Figure 3.12: Power spectra for 2CW interference environment.

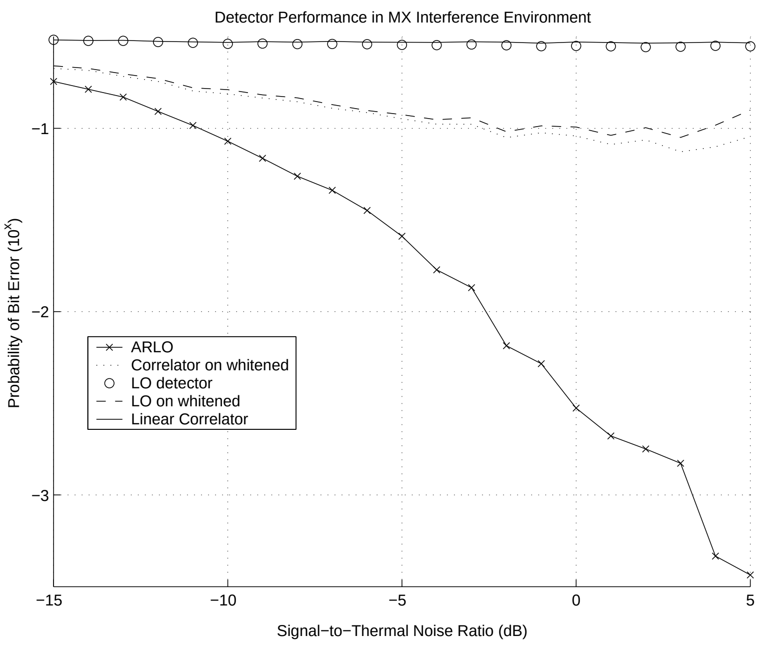

Figure 3.13: BER in MX interference environment.

over other standard detectors.

3.3 Performance Effects of Alternative Parameter Values

A number of parameters exist within the ARLO detector whose values are a major factor in its BER performance. Two such parameters were varied, to determine their specific effect on the ARLO detector. Bh, the pdf estimation parameter, defines the the number of symbols used in calculating the histogram used in estimating the pdf of the received signal. P, the AR-model parameter, defines the number of AR coefficients used in the AR-model. In addition, the ISR, a variable external to the communication system, was varied for completeness, as the power of an interference generator is an unknown and can be set to any practical level.

3.3.1 Bh: pdf Estimation Parameter

The first variable tested was Bh, the number of symbols used to calculate the histogram for pdf estimation. As will be discussed in chapter 4, Bh has a major effect on the simulation runtime. Therefore, a practical limit of Bh = 128 was

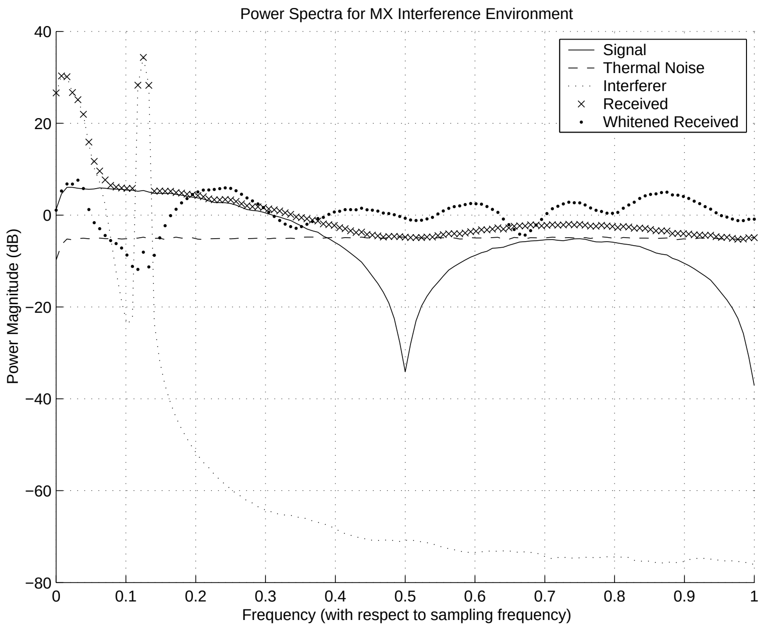

Figure 3.14: Power spectra for MX interference environment.

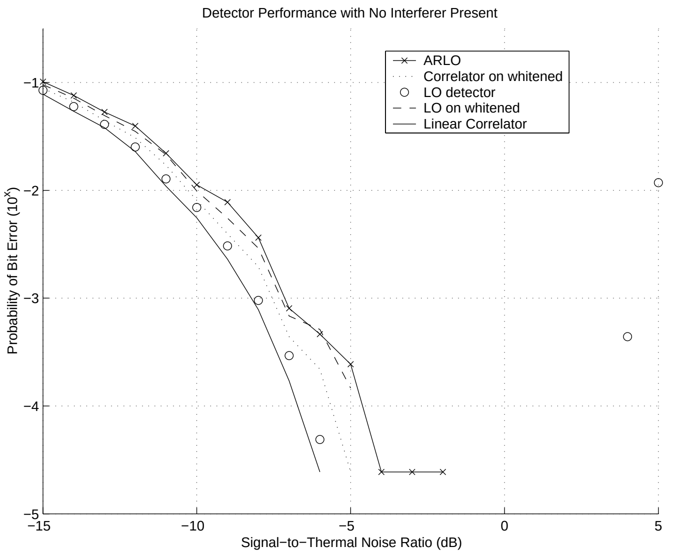

Figure 3.15: BER with no interferer present.

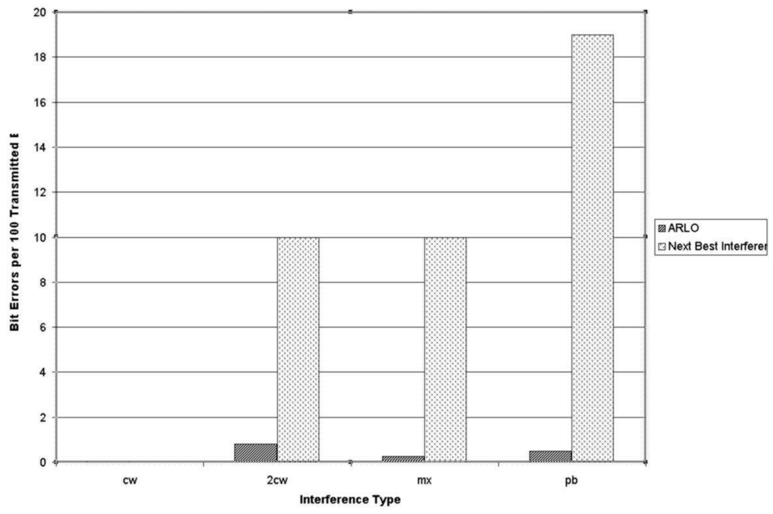

Figure 3.16: Number of bit errors at the receiver for every 100 bits transmitted, at 0 dB SNR.

chosen. A lower value was also decided upon, Bh = 64. Figure 3.17 shows the BER curve for a mixed interferer with Bh=128. Comparing this with the BER curve in Figure 3.18, a mixed interference environment with Bh=64, it is clear that a lower value of Bh results in a less-smooth plot. The BER varies more dramatically with a lower Bh, making its results less predictable. While the results are still good, the BER using Bh=64 never improves beyond 10^{-3}, implying that a lower Bh imposes a performance limit on the ARLO detector. This agrees with theory, as pdf estimation improves when calculated using a greater number of symbols. The better the noise pdf is estimated, the greater effect the ARLO detector can have in overcoming the noise. These results are corroborated in Figures 3.19 and 3.20, where a variable Bh is seen in a CW interference environment. The same unpredictability is seen, with a higher BER lower bound.

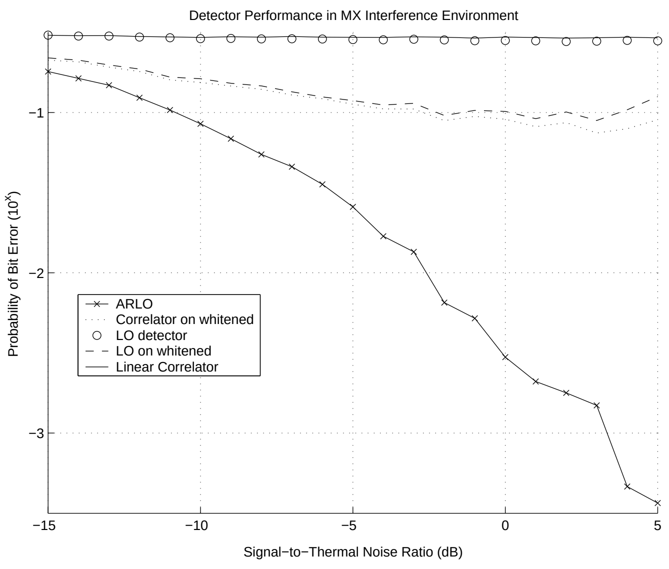

Figure 3.17: BER curve in Mixed interference environment with Bh = 128.

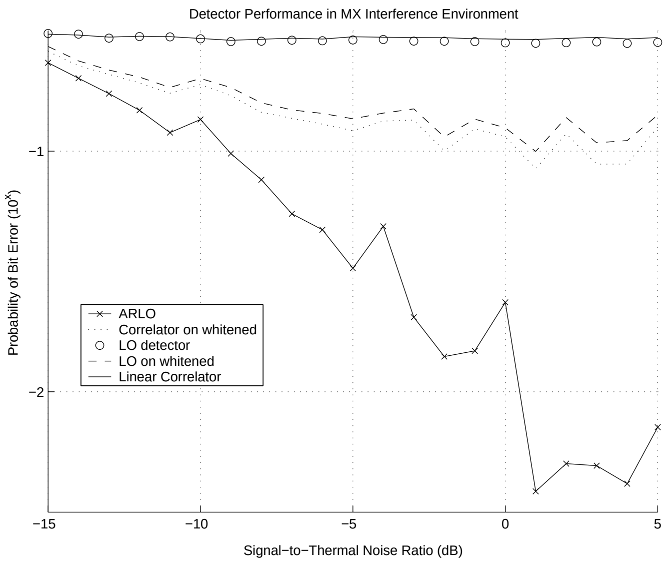

Figure 3.18: BER curve in Mixed interference environment with Bh = 64.

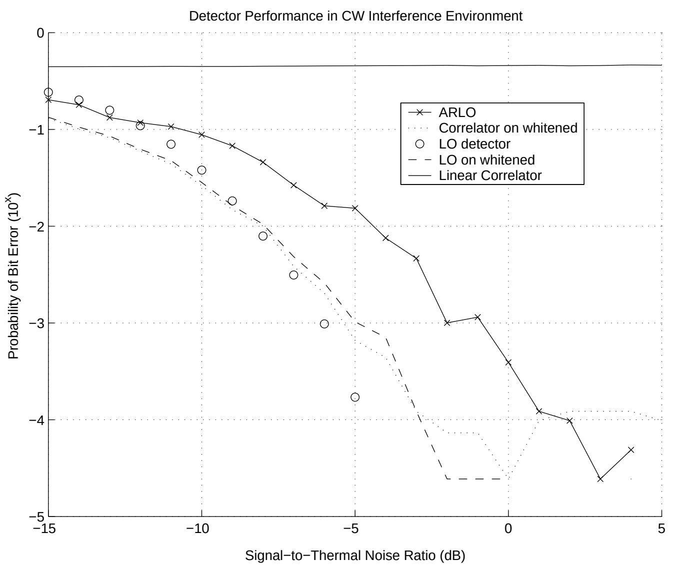

Figure 3.19: BER curve in CW interference environment with Bh = 128.

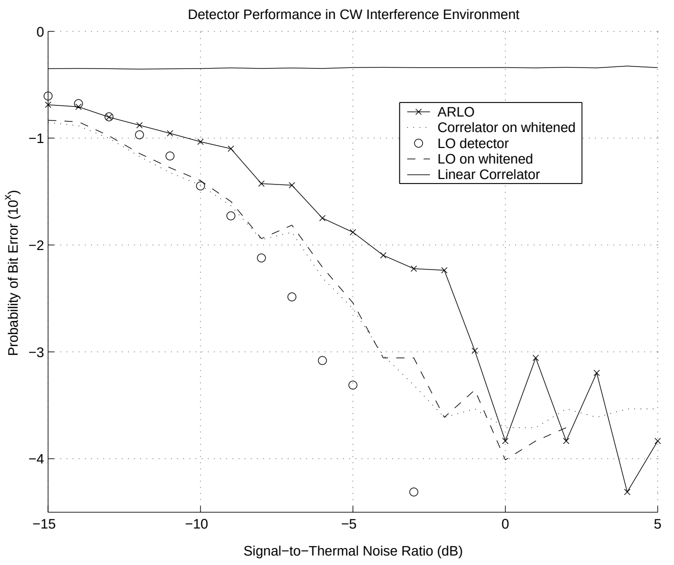

Figure 3.20: BER curve in CW interference environment with Bh = 64.

3.3.2 P: AR-model Parameter

The second variable tested was P, the number of AR coefficients used in the AR-model. P also has an effect on runtime, as discussed in chapter 4, so the values chosen for testing include P = [8, 16, 32]. Figures 3.21, 3.22, and 3.23 show BER curves with 2CW interferers and a variable P. It is interesting to note that as P decreases, the performance of ARLO increases. With P=32, the LOW and LCW actually outperform ARLO, whereas with P=8, the ARLO probability of bit error drops below 10^{-3}. The BER curves, with the CW interferer, in Figures 3.24, 3.25, and 3.26 concur. The ARLO BER with P=8 actually drops below 10^{-4}.

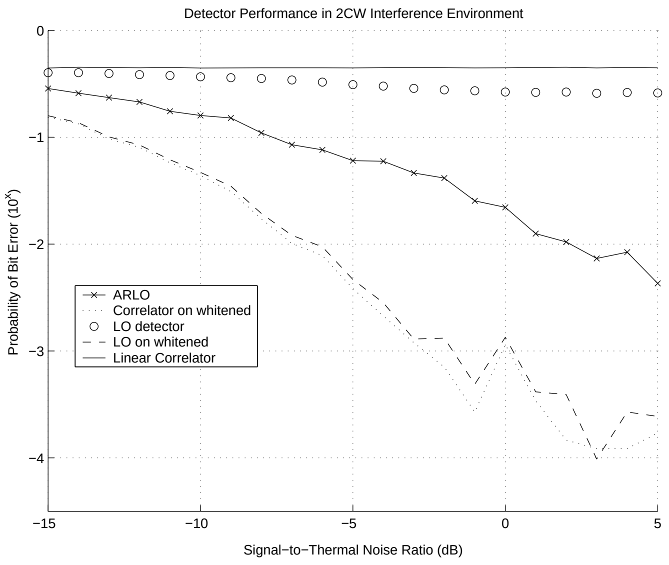

Figure 3.21: BER curve for 2CW interference environment with P=32.

3.3.3 ISR: Interference Parameter

Varying the ISR increases the practical significance of the simulations, as there is no guarantee how much interference power a communication system may face in the channel. Since varying the ISR has no effect on the simulation runtime, any practical values may be chosen. Tests were run using ISR = [20, 30, 40]. Figures 3.27, 3.28, and 3.29 show the BER performance in a PB interference environment with varying ISR. As expected, the performance of the detectors falls when more interference is present. ARLO continues to remain the best-performing detector. In Figures 3.30, 3.31, and 3.32, where the CW interferer is examined, detector performance also decreases as ISR increases. One thing to note is that with ISR=40 dBW, the ARLO detector begins outperforming two

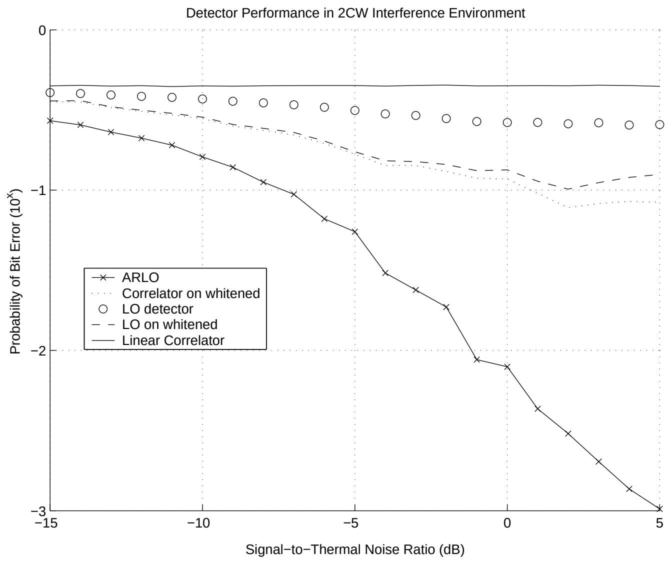

Figure 3.22: BER curve for 2CW interference environment with P=16.

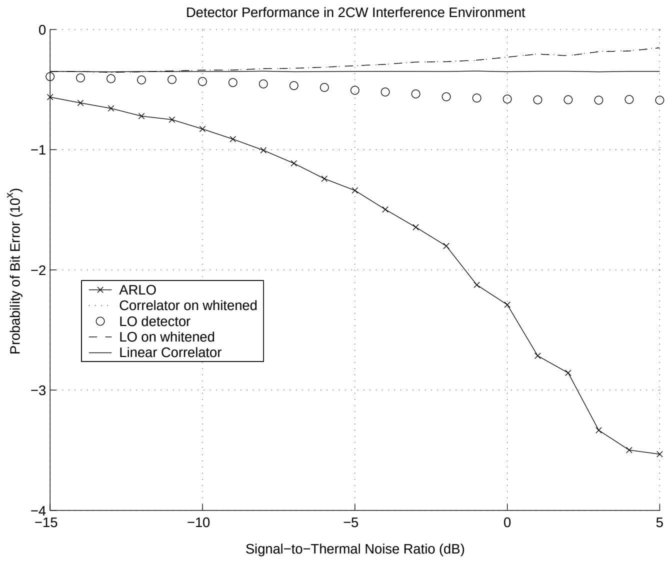

Figure 3.23: BER curve for 2CW interference environment with P=8.

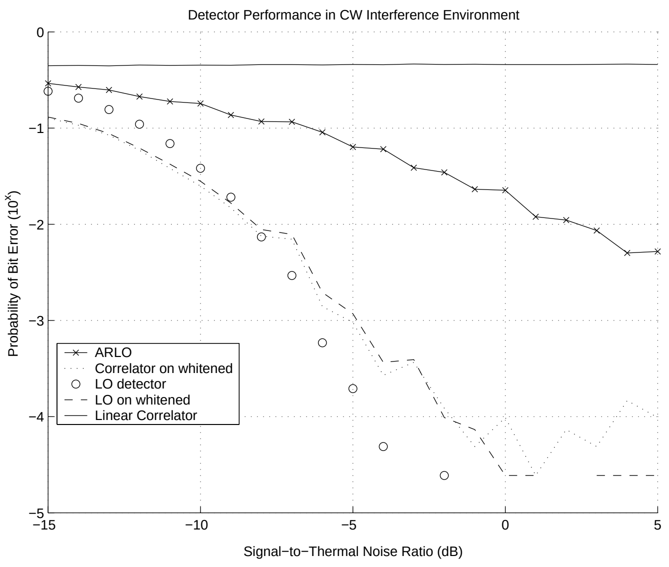

Figure 3.24: BER curve for CW interference environment with P=32.

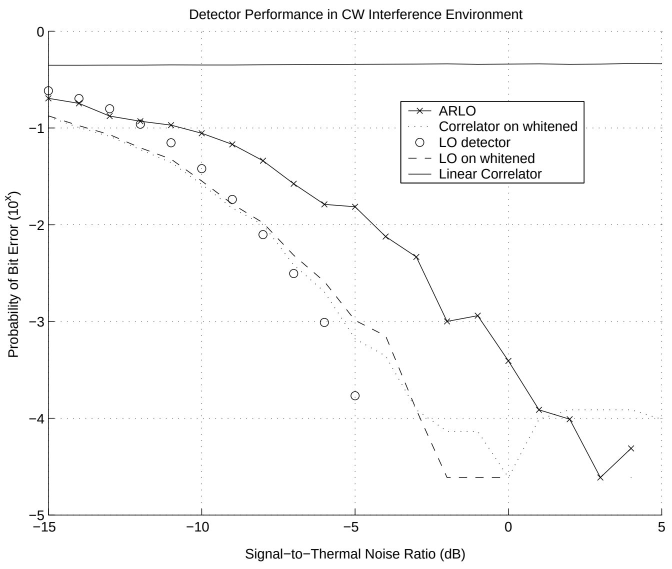

Figure 3.25: BER curve for CW interference environment with P=16.

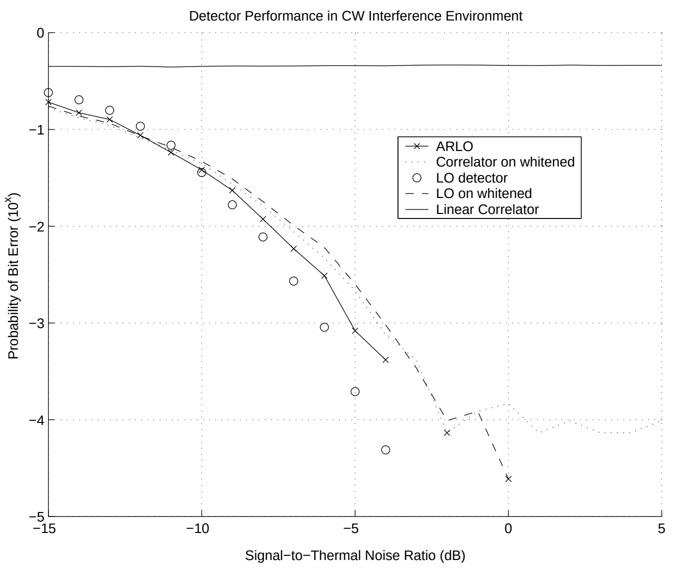

Figure 3.26: BER curve for CW interference environment with P=8.

other detectors. This shows the impressive robust nature of the ARLO detector, even under heavy interference.

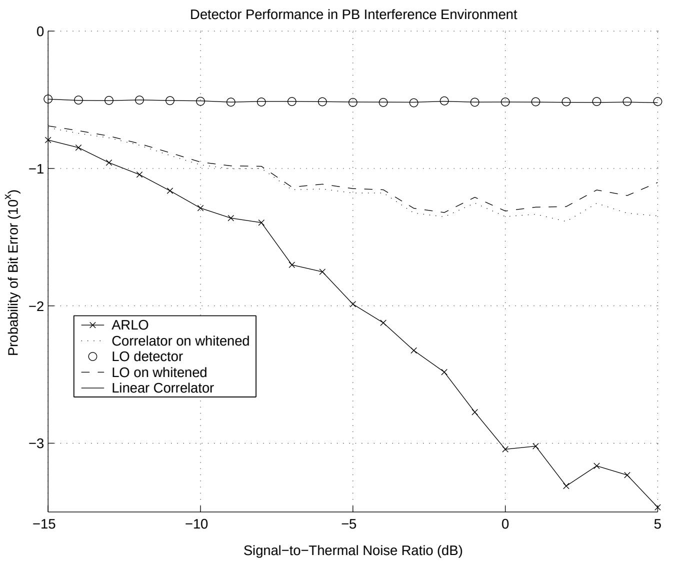

Figure 3.27: BER curve for PB interference environment with ISR=20 dbW.

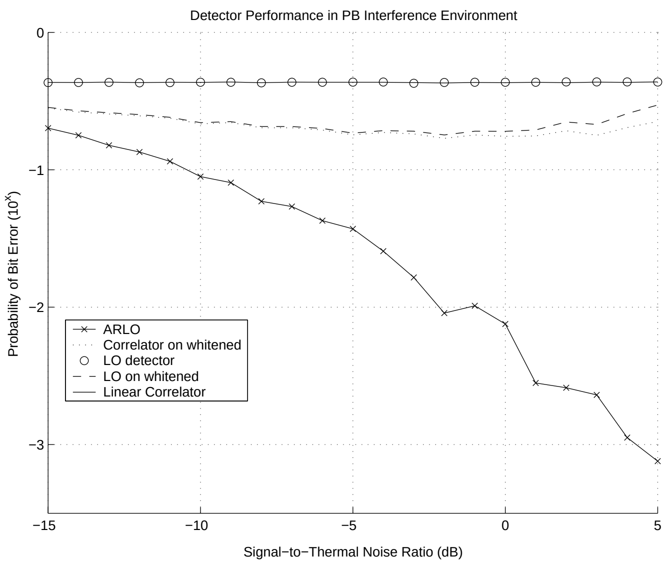

Figure 3.28: BER curve for PB interference environment with ISR=30 dbW.

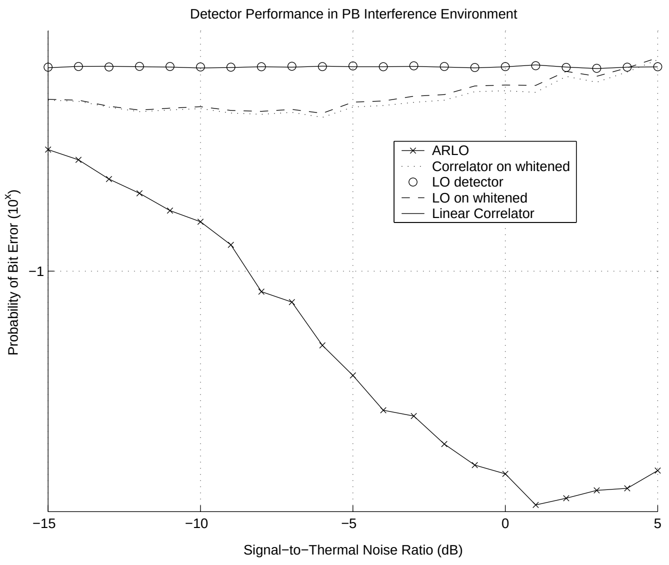

Figure 3.29: BER curve for PB interference environment with ISR=40 dbW.

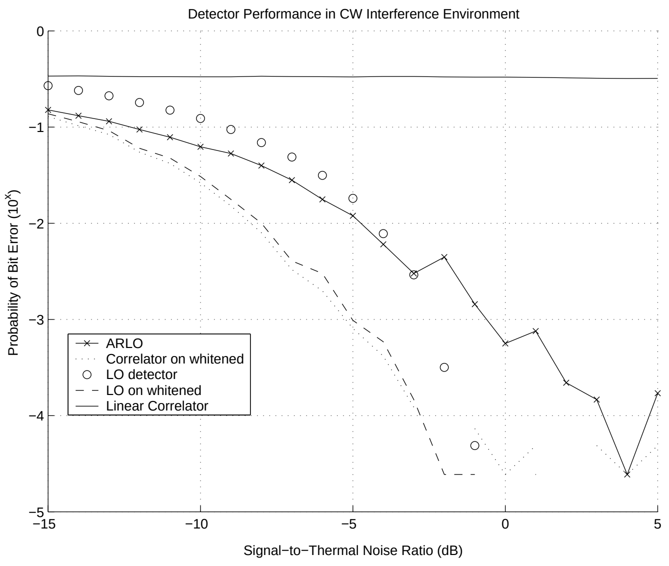

Figure 3.30: BER curve for CW interference environment with ISR=20 dbW.

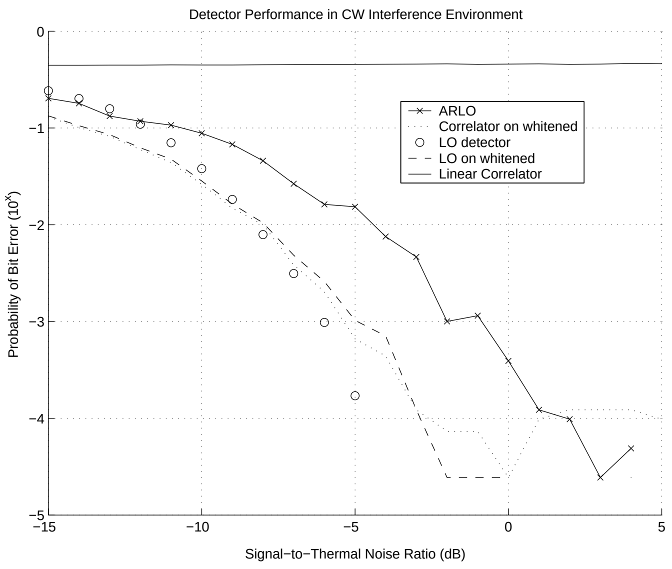

Figure 3.31: BER curve for CW interference environment with ISR=30 dbW.

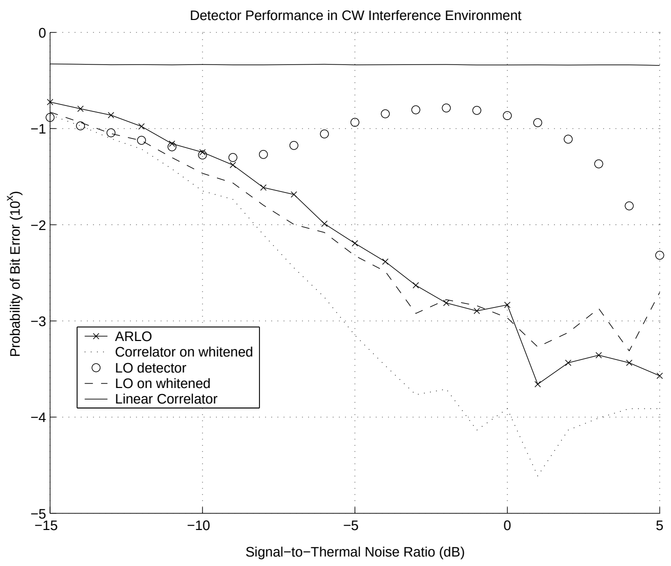

Figure 3.32: BER curve for CW interference environment with ISR=40 dbW.

3.4 Autoregressive Modeling

Two methods were examined in order to create an autoregressive model of the received signal vector. The *Modified Covariance Algorithm* (MCA) was initially chosen in this capacity because of its simplicity. The MCA generates a P^{th} order all-pole filter model to represent an input signal by minimizing the sum of the forward and backward prediction errors to create a lattice model of a random process. The forward prediction error is defined as,

$$e^{+}(n) = x(n) + \sum_{k=1}^{P} a(k)x(n-k), \tag{3.6}$$

and the backward prediction error is,

$$e^{-}(n) = x(n-P) + \sum_{k=1}^{P} a^{*}(k)x(n-P+k), \tag{3.7}$$

where a^* indicates conjugation, and P is the order of the all-pole filter. The sum to be minimized is then,

$$\varepsilon^{M} = \sum_{n=P}^{N} \left[\left|e^{+}(n)\right|^{2} + \left|e^{-}(n)\right|^{2} \right]. \tag{3.8}$$

To find the filter coefficients, $a(k)$, necessary to minimize ε^M, (3.8) is manipulated, with the result being the normal equations,

$$\begin{array}{c} \sum_{k=1}^{P} \left[r_x\left(l,k\right) + r_x\left(P-k,P-l\right)\right] a\left(k\right) = -\left[r_x\left(l,0\right) + r_x\left(P,P-l\right)\right] \\ l = 1,\ldots,P \end{array} , \tag{3.9}$$

where $r_x(l,k)$ is the auto-correlation of the process $x(n)$,

$$r_x(l,k) = \sum_{n=P}^{N} x(n-k)x^*(n-l). \tag{3.10}$$

Using (3.10) to obtain the auto-covariance sequences for the random process within (3.9) allows the normal equations to be solved to obtain the filter coefficients, $a\left(k\right)$.

The Burg method was also examined due to its performance characteristics. However, tests revealed no significant performance enhancement. Therefore, the MCA was chosen because of its simplicity.

Chapter 4

Simulation Conversion

The conversion of the simulation into a high-level language is a major component of this research project, for two reasons. First, it is expected that porting the simulation into a compiled language will increase the execution speed over the interpreted code used by MATLAB® . Second, with the ultimate goal of implementing the ARLO detector on a DSP, many utilities exist that can easily translate code written in certain high-level languages into DSP machine language. Therefore, the porting of the code is the first step towards a hardware implementation of the ARLO detector.

This chapter will thoroughly discuss the conversion process. The first section will detail the runtime of the MATLAB® simulation, including the effect that certain parameters have on execution time. The second section will discuss using the MATLAB® Compiler to port the simulation into *C*. The design and planning for the conversion is covered in the third section, including choice of language and platform options. The fourth section will discuss the functions requiring conversion, as well as some major obstacles faced in the conversion process. Finally, the fifth section discusses the results of the conversion, and compares the output of the translated code to that of the MATLAB® simulation.

4.1 Timing Analysis

The runtime of the MATLAB® simulation, defined as the elapsed time between the initial start time and the time the simulation finishes, is a major concern because it is the sole determining factor of whether the ARLO detector can be used in a real-time application, or whether its capabilities must be taken advantage of offline. The initial timing analysis was performed using old hardware, with the fastest system being a Pentium II 300 MHz, with 256 MB of RAM. However, more current hardware has been obtained, and a new timing analysis performed that agrees in relative terms with the initial analysis, but with faster overall execution speed. The new system is a dual-processor Pentium III XEON 600MHz, with 1 GB of RAM. The results detailed below were measured from this new system, using the default parameters as defined in Table 3.1, unless otherwise noted.

All of the timing results were very consistent, so two simulations were chosen as a representative sample, the CW and PB cases. The overall runtime for the two simulations are shown in Table 4.1, with a difference in overall runtime of 1 minute between the two simulations. Many events comprise the fifteen hour runtime of the simulation. These include creating a data stream to transmit, generating thermal background noise, generating the interferer, corrupting the signal in the channel, and performing detection using all five detectors. This being the case, the fifteen hour runtime is not, by itself, representative of the time needed for ARLO to perform.

Simulation Type	**Runtime (hrs)**
CW	15.1119
PB	15.1338

Table 4.1: Total runtime, in hours, of two representative simulations.

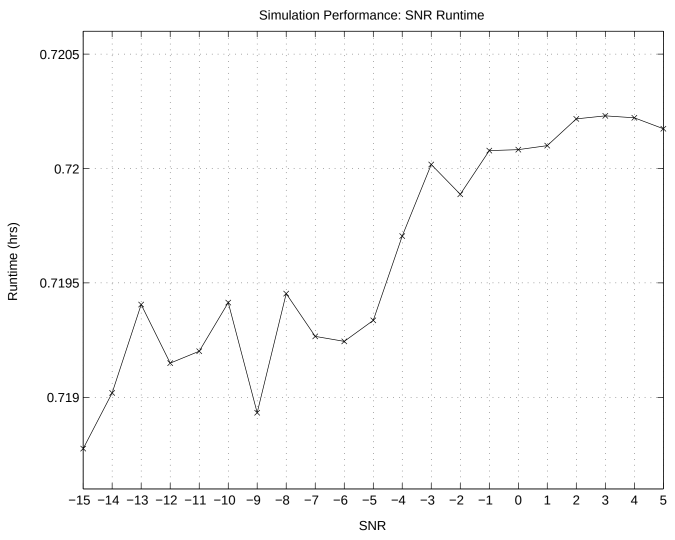

Figure 4.1: Runtime of simulation with CW interferer, over SNR.

To obtain a greater understanding of the runtime, Figures 4.1 and 4.2 show the runtime of each simulation, broken down by SNR iteration. The variability of the plots is misleading, however, as the scale of the ordinate is expanded. The runtime of each SNR ratio, as shown in Figure 4.1, varies from 0.7188 hours to 0.7202 hours, a negligible difference of 5 seconds. For analysis, the average time

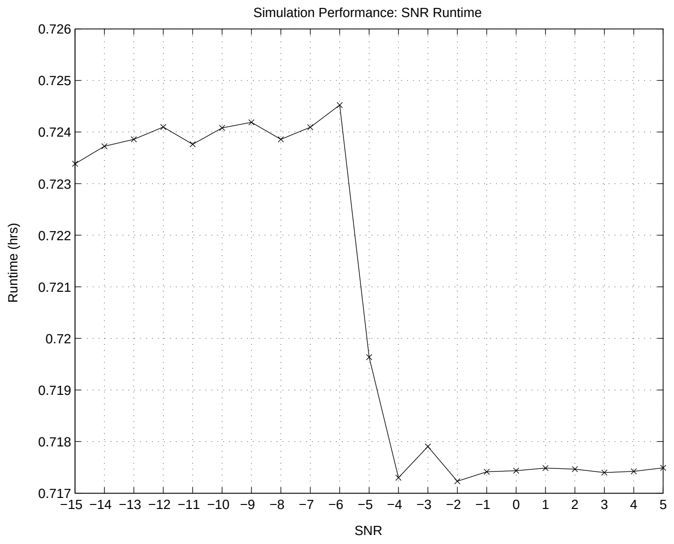

Figure 4.2: Runtime of simulation with PB interferer, over SNR.

per SNR iteration, for both simulations, is 0.72 hours, or 43.2 minutes. Each SNR iteration is composed of 10 *Monte Carlo* (MC) iterations. So the average MC iteration runtime is,

$$\frac{43.2\, min}{10\, iteations} = \frac{4.32\, min}{iteration}. \tag{4.1}$$

More detailed study needs to be performed to get exact values (see Chapter 5), but the ARLO detector, on a single data stream of 16,384 bits, requires approximately four minutes to completely detect the received data stream. While fast, ARLO is clearly not real-time capable in its current incarnation.

4.1.1 Simulation Runtime Effects of Alternate Parameter Values

As discussed in section 3.3, where changing certain communication system parameter values results in different BER performance results, changing these same parameters also often has an effect on the runtime of the simulation. Table 4.2 details the overall simulation runtime when changing the listed parameters. Changing the two internal parameters has a noticeable effect on simulation runtime. Halving Bh results in halving the simulation runtime, a clear 1:1 direct relationship. Assuming all else remains constant, the ARLO detector takes approximately two minutes to decode a 16,384 bit data stream with Bh=64. This is a great runtime performance improvement. But the drawback is that the BER performance degrades, as shown in section 3.3.1. Changing the value of

Parameters	CW Case (hrs)
Default (Bh=128, P=16, ISR=30 dB)	15.1119
Bh=64	7.53
P=8	13.6932
P=32	20.2771
ISR=20 dB	15.1015
ISR=40 dB	15.1235

Table 4.2: Simulation runtime using alternate parameter values.

P also has an effect on performance. Reducing P to 8 results in a 9.4% runtime performance increase, while increasing P to 32 results in a 34% runtime performance loss. Changing the ISR, a variable external to the communication system, has no effect on runtime, as expected.

4.2 The MATLAB® Compiler

As discussed in the introduction to this chapter on page 32, it is well-known that MATLAB® code, while convenient, is inherently slow because the MATLAB® program is an interpreter, which directly executes high-level language code without first compiling it into machine language. The runtime of the simulation can thus be greatly reduced if the mathematically intensive functions of the simulation execute in C or C++, languages whose code is compiled before execution.

The ideal solution is to recode the simulation by hand into C or C++, thereby optimizing it as much as possible. However, in order to get results in the short term, an automatic, machine-generated translation is a good alternative. The MATLAB® Compiler, in conjunction with a standard compiler such as *Microsoft Visual C++*, translates most MATLAB® code into C. It then compiles the C code into a MEX-file, a binary format directly accessible from an M-file (the standard file used for MATLAB® code), i.e. the M-file syntax to call a function located in another file is the same whether the target file is a MEX-file or an M-file.

The major benefit of using MEX-files is its handling of loops. C and C++ are extremely efficient in executing any type of loop, whereas MATLAB® is notoriously poor [6, pp. 4], simply due to its status as an interpreter. This, in itself, is not enough to give a dramatic improvement in simulation runtime because the current code uses few large loops. However, the MATLAB® Compiler provides numerous compilation options that can further enhance the runtime. They allow the compiler to generate simpler data types, resulting in simpler, and thus faster, code.

Two of the options available are particularly applicable, and provide a huge performance increase when used in combination. The first is the assertion that all floating-point variables contain real numbers. This allows the compiler to remove all code whose purpose is to handle complex numbers, thereby reducing the complexity of the code and, thus, the runtime. The second option generates code that restricts matrices from growing beyond their initial size and, consequently, never checks a matrices' bounds. Using these two assertions to create MEX-files, the simulation runtime decreases by roughly a factor of two.

One problem arose while running simulations using the MEX-files. Depending on the specific parameters of the simulation, the pdf estimation sometimes resulted in complex AR coefficients. This caused the simulation to stop executing, with an appropriate error message. Recompiling the MEX-files without the real number assertion will remove this error. However, that assertion is the primary factor in the speed increase of the simulation. Thus, removing this assertion will likely negate much of the performance improvement [6, pp. 15].

As promising as using the MATLAB® Compiler appears, we receive no performance benefit from it if the speed-increasing assertions cannot be used. Therefore, to convert the simulation to a high-level language, porting it by hand is necessary.

4.3 Software Design

As in all programming projects, detailed planning must precede the actual coding process. The first important issue is to decide which programming language will be optimal for this project. Both C and C++ were considered due to their common usage as DSP languages. Since C supports structured programming, and this method provides a good representation of the simulation, our program will not benefit from the object-oriented extensions that C++ provides. Therefore, C was chosen as the programming language.

The next important issue is the platform on which the simulation will be developed. One option is Linux, which provides solid stability, as well as improved performance and memory management over Microsoft Windows® . Windows, however, is more accessible to the people who will be performing the code translation, and it provides an enhanced development and debugging environment in Microsoft Visual C++ 6.0® . Therefore, Windows was chosen as the development platform. However, in order not to restrict future development, we decided to write the code using only standard ANSI C features, which should make it highly portable to Linux, if desired.

The last issue is to decide which data type would best represent the signal samples used in the simulation. MIX Software's C/Math Toolchest©, which provides vector and matrix math libraries, can be compiled to use either floating point precision or double precision. Double precision is clearly superior in its ability to provide a very accurate representation of the signals. But since MATLAB® uses double precision, and numerous problems have arisen regarding memory usage in past simulations, floating point precision was chosen to determine if the added precision is necessary or if it poses an inefficient use of computing resources.

4.3.1 Simulation Block Diagram

To convert the simulation into C, it is imperative to understand the communication system being simulated. An excellent way of accomplishing this is to study a visual representation of the communication system. The block diagram, shown in Figure 4.3, was created for this purpose.

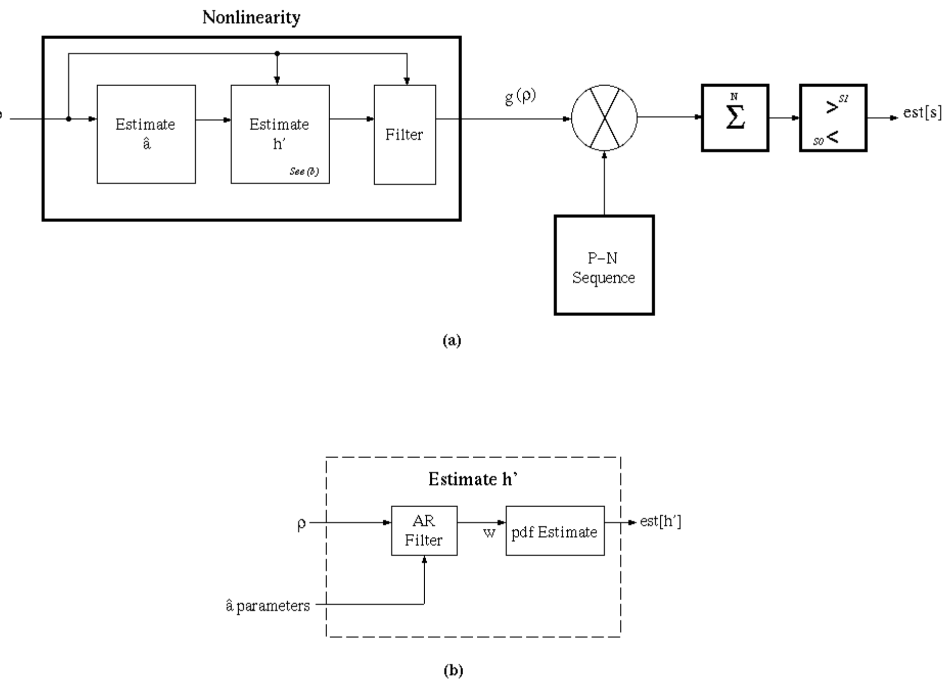

Figure 4.3: (a) ARLO system block diagram. (b) Expanded view of *Estimate h'* block.

Figure 4.3a provides the overall signal flow of the receiver. Figure 4.3b illustrates the detail of the nonlinearity for estimation of the function *h'* (to be described later). In Figure 4.3a, ρ is the received signal vector of length N, which is composed of both the signal sent by the transmitter and the noise present in the channel. The receiver must first calculate the LO nonlinearity, $g(\rho)$, which is individually calculated for each incoming symbol in ρ, each of which is composed of N_b samples per symbol. The next step is to de-spread the "chipped" data by using the ***pseudo-noise*** (PN) sequence. Finally, an estimate of the transmitted signal, est[s], is computed by summing the de-spread samples of the symbols and thresholding.

The receiver output, est[s], is highly dependent on the nonlinearity computation. This computation is a multi-step process. The first step in calculating the LO nonlinearity is the computation of the P^{th}–order AR model coefficients,

$\{\hat{a}_i\}$, $i = 0, 1, 2, \ldots, P$. To date, the *modified covariance algorithm* (MCA) is used for this purpose. The second step is to estimate the pdf of the noise. This is currently accomplished via a histogram approximation method. This process is shown in detail in Figure 4.3b. The received signal, ρ, passes through the AR filter, with coefficients $\{\hat{a}_i\}$, as shown in Figure 4.3a, to create a whitened signal vector, $\mathbf{w}$. The histogram of $\mathbf{w}$ is then computed. The result represents the pdf of the whitened received signal

The pdf of the noise component of $\mathbf{w}$ can be written as,

$$\begin{array}{rcl} f_n(\eta) & = & f_n(\eta_1, \eta_2, \ldots, \eta_N) \\ & = & \prod_{i=1}^{N} f_n(\eta_1 \mid \eta_{i-1}, \ldots, \eta_{i-P}) \end{array}, \tag{4.2}$$

where

$$f_{n_i}(\eta_i \mid \eta_{i-1}, \ldots, \eta_{i-P}) = \left\{ \begin{array}{l} f_{n_i}(\eta_i), \text{ for } i = 1 \\ f_{n_i}(\eta_1 \mid \eta_{i-1}, \ldots, \eta_1), \\ \quad \text{for } i = 2, \ldots, P \end{array} \right. . \tag{4.3}$$

This can be rewritten as,

$$f_n(\eta) = \prod_{i=1}^{N} f_w \left(-\sum_{j=0}^{P} a_j \eta_{i-j} \right), \tag{4.4}$$

where $f_w(\omega)$ is the pdf of the white noise process and $a_0 = -1$. The result of this block, as estimate of h', est[h'], is the derivative of the natural log of the pdf of $\mathbf{w}$, that is,

$$\text{est}[h'(\omega)] = \frac{\mathrm{d}}{\mathrm{d}\omega} \{\ln[f_w(\omega)]\}. \tag{4.5}$$

The next, and final, step in calculating $g(\rho)$ is to use ρ and h' in the filtering process,

$$g_i(v) = \sum_{l=0}^{\min(P,N-i)} a_l h' \left(-\sum_{j=0}^{P} a_j v_{i+l-j} \right), \text{ for } i = 1, \ldots, N, \tag{4.6}$$

where

$$v_i = \left\{ \begin{array}{l} 0, \text{ for } i \le 0 \\ \rho_i, \text{ for } i \in [1, N] \end{array} \right. . \tag{4.7}$$

4.4 Code Translation

The simulation is composed of two distinct sections. The first section implements the direct sequence spread spectrum generator, the communication channel, including a thermal noise generator and interference generator, and the linear correlator receiver. The second part of the simulation contains the nonlinear LO receivers. The conversion into C was performed along these lines.

4.4.1 Function List

Since the research team decided to port the simulation, based on the MATLAB® simulation, into C, the functional breakdown required in C will be very similar to that used in MATLAB® . The list of functions in the completed C code include the following:

- ar_histo.c: Estimates the pdf of a white noise process, without assuming any underlying structure for the white noise pdf.
- ar_theory.c: Estimates the pdf of a white noise process, assuming that the pdf of the white noise has a known underlying structure, i.e. Gaussian, Poisson, or Laplacian.
- contwave.c: Creates CW interferer.
- conv.c: Performs convolution of two vectors of polynomial coefficients.
- decision.c: Makes decision on received symbol.
- dsss.c: Generates direct-sequence spread-spectrum signal.
- fftshift.c: Swaps left and right halves of an input vector, placing the DC (frequency=0) component in the middle of the spectrum.
- filter.c: Filters a data stream based on input filter coefficients.
- fliplr.c: Flips the columns of a matrix in the left-right direction about a vertical axis.
- flipud.c: Flips the rows of a matrix in the up-down direction about a horizontal axis.
- hist.c: Creates histogram of a data stream.
- modcov.c: Finds P^{th}–order all-pole model for a signal using the MCA method.
- nlp.c: Calculates the LO nonlinearity, $g(\rho)$.
- read_parms.c: Inputs the simulation parameters from a configuration file.
- receiver.c: Performs synchronous reception of a transmitted signal.
- reshape.c: Reshapes a matrix.
- simulation.c: Main program simulation file.
- th_inter.c: Creates a theoretical PB interferer.
- toeplitz.c: Creates a Toeplitz matrix from two input vectors.

The list of header files include:

- mathlib.h: Header file for C/Math Toolchest® .
- function_list.h: Function prototypes.
- sim_parameters.h: Simulation global variable declarations.
- sim_parameters_extern.h: Simulation global variable declarations, externalized.

The complete source code for the simulation can be found in Appendix A.

4.4.2 Plotting Scripts

After debugging was complete, the research team required a method to plot the results of the simulation, in order to compare its output with that of MATLAB® . With the final goal of running the ARLO detector on a DSP, it is unnecessary to write a routine in C which will handle BER plotting, especially since graphical coding in C would add a level of complexity to the conversion attempt that is not required. Since MATLAB® is particularly suited for graphical plotting, a script was written in MATLAB® that can read in the output of a C simulation from a file and graph the BER and spectral plots. These plots can then be compared with those from a MATLAB® simulation, for testing purposes. The source code for this script can be seen in Appendix B.

4.5 Conversion Results

The results of the conversion were highly successful. Comparisons of BER performance, using identical system parameters, of the C and MATLAB® simulations are shown in Figures 4.4, 4.5, 4.6, and 4.7. From these figures, it is clear that the simulation has been successfully ported to C, as the BER curves are nearly identical.

The power spectra curve of the PB scenario, shown in Figures 4.8 and 4.9, are also nearly identical, therefore showing that the conversion was a success.

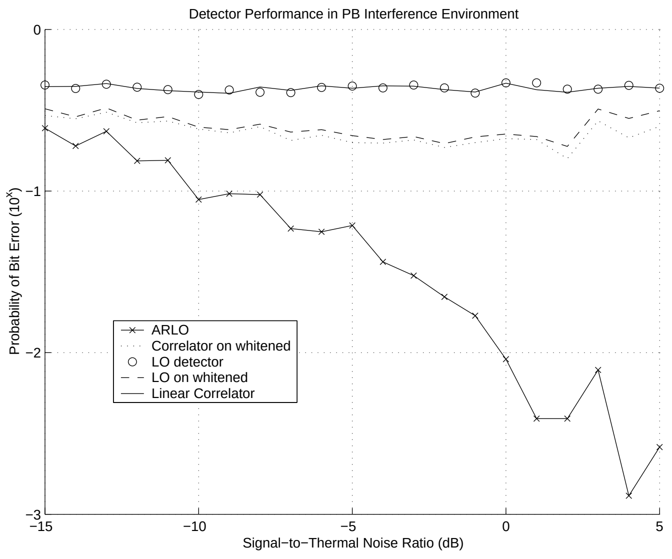

Figure 4.4: C Simulation: BER curve in PB interference environment.

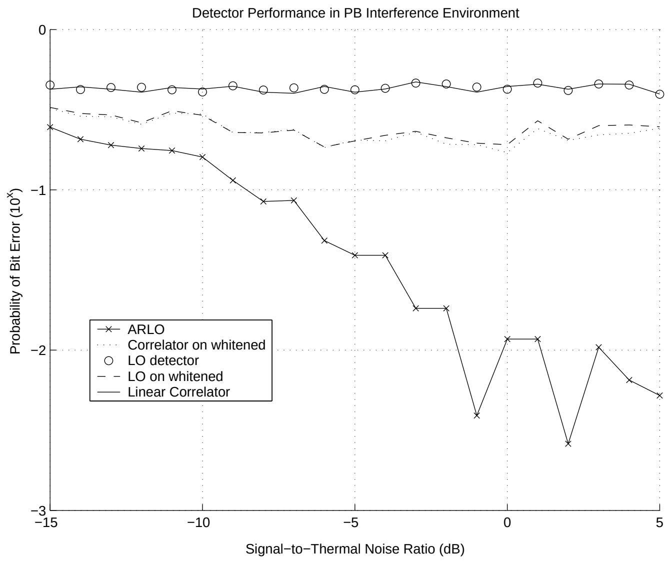

Figure 4.5: MATLAB® Simulation: BER curve in PB interference environment.

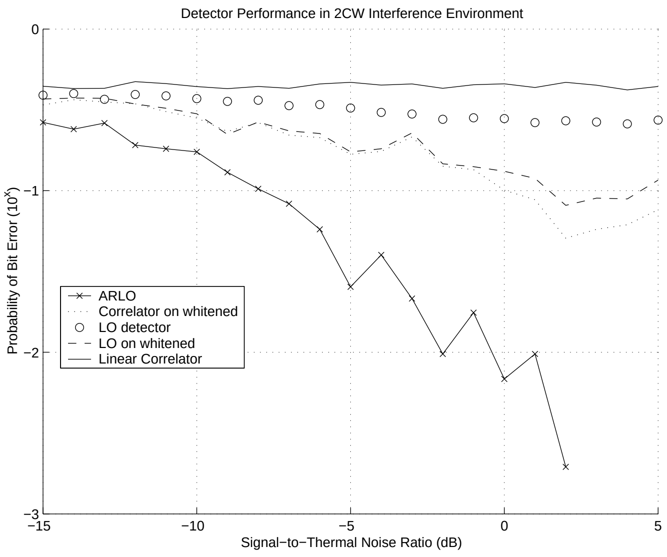

Figure 4.6: C Simulation: BER curve in 2CW interference environment.

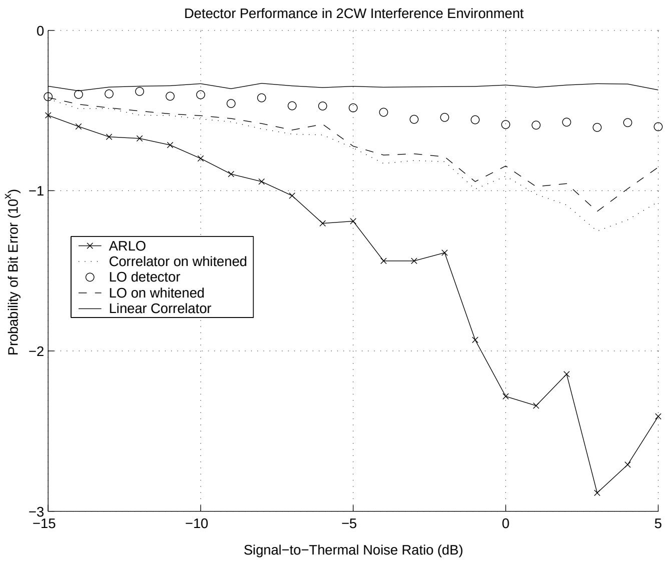

Figure 4.7: MATLAB® Simulation: BER curve in 2CW interference environment.

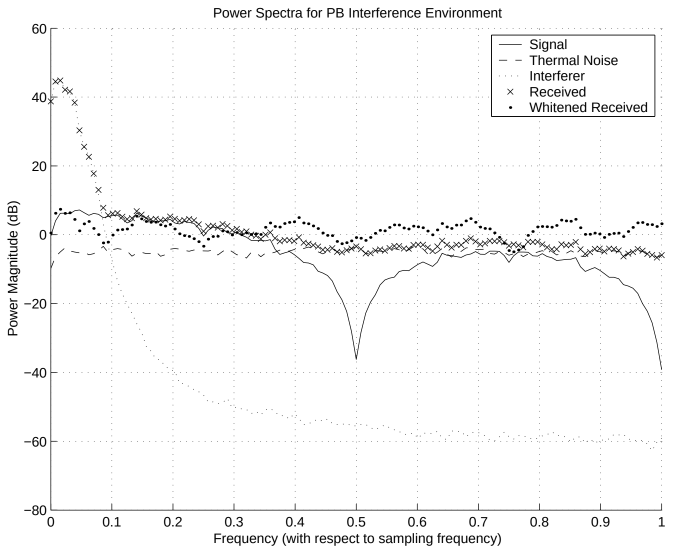

Figure 4.8: C Simulation: Power spectra of PB interference environment.

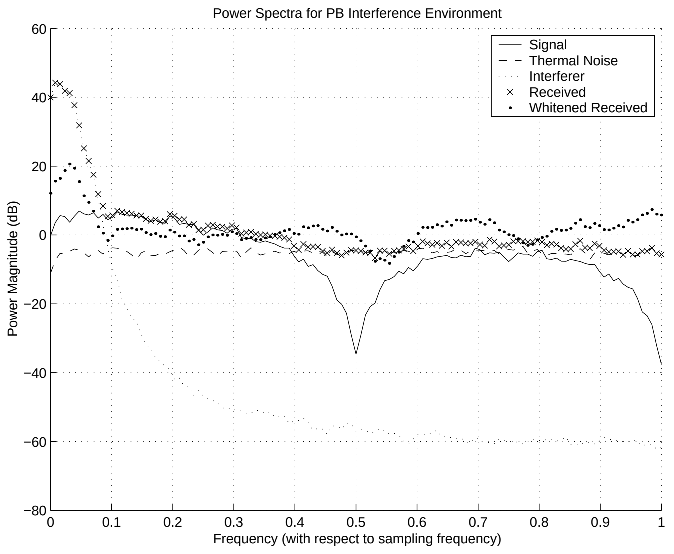

Figure 4.9: MATLAB® Simulation: Power spectra of PB interference environment.

Chapter 5

Summary and Future Research

In summary, the work performed by the research team was highly successful. The performance tests expanded on previous research to prove the efficacy of the ARLO detector in intense interference environments, as well as its superior BER performance in most every scenario over the other popular detector algorithms. Simulation results also showed that, in a non-hostile channel with zero interference, the performance differential between ARLO and the optimum linear correlator were negligible.

The conversion of the simulation into C was also a major milestone for the research team. Intelligent choices were decided upon with regard to the fundamentals of the conversion, including its platform, chosen programming language, and library selection. The conversion was completed and thoroughly tested to be equivalent to its MATLAB® predecessor.

Future work is needed to bring the ARLO detector into service. The C simulation performs at a much slower rate than the MATLAB® version. Since the C version is necessary for downloading the detector onto a *digital signal processor* (DSP), a detailed study of the required operating time of each module of the simulation is required to examine the feasibility of the ARLO detector for real-time applications. To accomplish these goals, the proposed research tasks include the following:

- Perform a detailed study and comparison of the runtime of the *C* and MATLAB® codes
- Optimize the *C* code to achieve maximum detector runtime performance
- Research other matrix libraries for decreased runtime
- Study the advantages and disadvantages of adapting the *C* code to execute on a Linux-based system

- Convert the *C* code for use on a Linux-based system
- Continue research on Parzen's estimator, and compare its performance with that of the histogram method.

5.1 Future Research Details

5.1.1 Runtime Study

The first area of future research is to perform a detailed timing analysis of the *C-coded simulation* (CCS). This includes gathering runtime data for each individual module of the CCS, comparing it to the timing results previously obtained from the *MATLAB®* *-coded simulation* (MCS), and determining which processes are consuming the most execution time. The most time-consuming modules will be examined more closely to ascertain the reasons behind this performance lag. Parameter variation, and its effect on runtime, will also be examined through structured simulations and timing comparisons.

5.1.2 Code Optimization

The current execution speed of the CCS is below expectations due to the research team's previous focus on obtaining total functionality at the expense of other factors. For effective, real-time application, it is imperative to increase execution speed. One method to accomplish this is to optimize the code itself, focusing primarily on modules exhibiting large execution time as measured in the runtime study (see 5.1.1). In addition, the compilers and linkers used in building this application have a great many optimization options to exploit for increased performance of the CCS. The simulation can also be adapted to run without including debugging information in the compiled executable, which will potentially result in a noticeable performance increase.

5.1.3 Matrix Libraries

To provide standard matrix and vector operations and manipulations, the CCS makes use of the *C/Math Toolchest*™code library. This code library, while inexpensive, easy to use, and fairly complete, is not optimized for current processors. Another area of future research is, therefore, to locate and study other matrix libraries, both open-source and commercial, and determine their suitability for the CCS. If a library is found that can potentially enhance detector performance, the research team will use its extensive programming experience to adapt the CCS to use the new library. The team will then perform a new detailed timing analysis to determine the exact performance benefits reaped from its use.

5.1.4 Platform Migration

The CCS was coded using *Microsoft Visual C++ 6.0*, running under *Microsoft Windows 2000.* It was written using standard ANSI C (no Microsoft extensions to *C* were used) to retain maximum portability. The proposed research includes examining whether CCS performance under Linux using a native *C* compiler, such as the *GNU Compiler Collection* (GCC), may give improved performance over the Microsoft-centric implementation. In the event that it does, the research team will migrate the code to Linux to harness the performance benefits of this operating system.

5.1.5 Parzen's Estimator

Further research is needed in order to determine if this new method definitely outperforms the histogram method. The performance time, input data size requirements, as well as the diversity of each method must be more closely examined. Future research may also include testing the performance of the Parzen estimator when alternative RBFs are used, as well as possibly researching other pdf estimation methods.

Appendix A

Simulation Source Code

A.1 Header Files

A.1.1 Function_list.h

```
/* Function prototypes */
void ar_histo (Real_Vector stimulus, unsigned stimulus_length, Real_Vector a,
                                        unsigned a_length, Real_Vector h, Real_Vector bin_width,
                                        Real_Vector mini);
   /* bin_width & mini are vectors to enable pass-by-reference */
void ar_theory (int iidparms[], char iidtype[], int ISR, Real K, Real T,
                                         Real_Vector h);
void contwave (Real_Vector interferer);
void conv (Real_Vector v1, int length1, Real_Vector v2, int length2,
                                Real_Vector output);
void decision (Real_Vector pn, Real_Vector g, Real_Vector decision);
void dsss (Real_Vector ref, int amp, Real_Vector coded_data,
                                Real_Vector chips);
void fftshift (Complex_Vector v1, unsigned int length1);
void filter (Real_Vector b, int M, Real_Vector a, int N,
                                      Real_Vector data_in, int input_length, Real_Vector y);
Real_Matrix fliplr (Real_Matrix M, unsigned m, unsigned n);
Real_Matrix flipud (Real_Matrix M, unsigned m, unsigned n);
void hist (Real_Vector signal, unsigned signal_length, unsigned nbins,
                                Real_Vector freq, Real_Vector bincenter);
void modcov (Real_Vector x, unsigned x_length, unsigned p,
                                      Real_Vector ARparm, Real error);
Real_Vector nlp (Real_Vector r, unsigned int r_length, Real_Vector mini,
                                               Real_Vector h, unsigned int h_length, Real_Vector bw,
                                               Real_Vector a, unsigned int a_length,
                                               unsigned int g_length);
int read_parms (char *filename);
void receiver (Real_Vector signal, Real_Vector noise, char *limit,
                                      Real_Vector rcvd_tr);
Real_Matrix reshape (Real_Matrix M, unsigned m1, unsigned n1, unsigned m2,
                                                        unsigned n2);
void th_inter (Real_Vector interferer);
Real_Matrix toeplitz (Real_Vector col1, Real_Vector row1, unsigned int nc1,
                                                              unsigned int nr1);
```

A.1.2 Sim_parameters.h

```
/* Simulation Parameters
  *
  *
  * By: Michael Banys
  *
  * January 26, 2000
  */
int
                 /* Book keeping parameters */
   it, /* Number of Monte Carlo iterations */
   Bh, /* Total number of symbols to calculate histogram */
   K, /* Data points to calculate */
   T, /* Range of support for pdf */
   z, /* Multiplier for determining number of transmitted symbols */
```

```
                /* Spread spectrum parameters */
   Nc, /* Samples per chip */
   PG, /* Processing gain (chips per bit) */
   Nb, /* Samples per symbol (assume symbol==bit) */
   B, /* Symbols transmitted/received and sent to BER tester */
   Ntot, /* Total number of samples in run */

                /* AR Model parameters */
   Bar, /* Total number of symbols to estimate AR parms */
   Bg, /* Total number of symbols to calculate nlp */

                /* Signal parameters & Thermal Noise */
   sigmat, /* Standard deviation of thermal noise */
   sigmat2, /* Variance of thermal noise */
   Eb_min, /* Minimum value of thermal SNR in dB */
   Eb_max, /* Maximum value of thermal SNR in dB */

                /* Correlated interferer parameters */
   ISR, /* Interferer-to-Signal ratio (dB) */

                /* Continuous Wave Interference */
   I, /* Number of CW interferers: 1 or 2 */
   P, /* Number of AR filter coefficients */

                /* Wide-band filtered white-noise jammers */
   L, /* ... for a filter of order 2^L */
   iidparms[2], /* Parameter vector for iidtype */

                /* Mixed Jammer */
   ISRcw, /* Divide up interferer power */
   ISRpb;
float
                /* Continuous Wave Jammer */
   freq[2], /* CW Interferer frequency(s) */

                /* Wide-band filtered white-noise jammers */
   polynomial[2];                                        /* The pole of the low-pass filter */
char
                /* AR Model Parameters */
   ar_method[5],                                 /* Coefficient estimation method
                                                'mcov' = Modified Covariance */

                /* Signal Parameters and Thermal Noise */
   limit[5],                                                     /* Limiter on noise:
                                                                   'hard' - noise blanker
                                                                   'soft' - soft limiter */

                /* Correlated interferer parameters */
   Itype[3],                                                     /* Interference type:
                                                                    'cw' - Continuous Wave
                                                                    'pb' - Partial Band
                                                                    'mx' - Mixed Partial Band and CW */

                /* Wideband filtered white-noise jammer */
   iidtype[3];                                           /* The type of iid noise sequence */
Real_Vector  Eb_sigma, poly, a_t;
```

A.1.3 Sim_parameters_extern.h

```
/* Simulation Parameters (extern)
  *
  *
  * By: Michael Banys
  *
   * January 26, 2000
  */
extern int
                /* Book keeping parameters */
   it, /* Number of Monte Carlo iterations */
   Bh, /* Total number of symbols to calculate histogram */
   K, /* Data points to calculate */
   T, /* Range of support for pdf */
   z, /* Multiplier for determining number of transmitted symbols */

                /* Spread spectrum parameters */
   Nc, /* Samples per chip */
   PG, /* Processing gain (chips per bit) */
   Nb, /* Samples per symbol (assume symbol==bit) */
   B, /* Symbols transmitted/received and sent to BER tester */
   Ntot, /* Total number of samples in run */

                /* AR Model parameters */
   Bar, /* Total number of symbols to estimate AR parms */
   Bg, /* Total number of symbols to calculate nlp */

                /* Signal parameters & Thermal Noise */
   sigmat, /* Standard deviation of thermal noise */
   sigmat2, /* Variance of thermal noise */
   Eb_min, /* Minimum value of thermal SNR in dB */
   Eb_max, /* Maximum value of thermal SNR in dB */

                /* Correlated interferer parameters */
   ISR, /* Interferer-to-Signal ratio (dB) */

                /* Continuous Wave Interference */
   I, /* Number of CW interferers: 1 or 2 */
   P, /* Number of AR filter coefficients */

                /* Wide-band filtered white-noise jammers */
   L, /* ... for a filter of order 2^L */
   iidparms[2], /* Parameter vector for iidtype */

                /* Mixed Jammer */
   ISRcw, /* Divide up interferer power */
   ISRpb;
extern float
                /* Continuous Wave Jammer */
   freq[2], /* CW Interferer frequency(s) */

                /* Wide-band filtered white-noise jammers */
   polynomial[2];                                   /* The pole of the low-pass filter */
extern char
                /* AR Model Parameters */
   ar_method[5],                            /* Coefficient estimation method
                                               'mcov' = Modified Covariance */

                /* Signal Parameters and Thermal Noise */
   limit[5],                                                /* Limiter on noise:
                                                               'hard' - noise blanker
                                                               'soft' - soft limiter */

                /* Correlated interferer parameters */
   Itype[3],                                                /* Interference type:
                                                               'cw' - Continuous Wave
                                                               'pb' - Partial Band
                                                               'mx' - Mixed Partial Band and CW */

                /* Wideband filtered white-noise jammer */
   iidtype[3];                                      /* The type of iid noise sequence */
extern Real_Vector  Eb_sigma, poly, a_t;
```

A.2 Program Code Files

A.2.1 Simulation.c

```
/* Program Simulation main file
  *
  *
  * By: Michael Banys
  */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "mathlib.h"
#include "sim_parameters.h"
#include "function_list.h"
void
main (int argc, char *argv[])
{
/* PROGRAM DECLARATIONS */
   int i, j,                                                /* Counter variable */
     saveData,                                      /* To control saving data only once */
     where_running,                         /* To determine where to save data */
     snr,                                                           /* SNR iteration */
     pe_avg,                                                /* Monte Carlo iteration */
     amp,                                                           /* Voltage amplitude of signal */
     Nsamh,                                                 /* Number of samples in each histogram = Bh*Nb */
     histonum,                                      /* Process Bh*Nb samples at a time for a histogram (loop var) */
     symcnt;                                                /* Counter for each symbol in AR section */
   Real sym_amp,                            /* Symbol amplitude assuming chip-matched filtering */
     num_errs_lc,                                   /* Number of errors for linear correlator */
     num_errs_wc,                                   /* Number of errors for lc on whitened */
     num_errs_lo,                                   /* Number of errors for lo */
     num_errs_lw,                                   /* Number of errors for lw */
     num_errs_ar,                                   /* Number of errors for ar */
     error;                                                 /* Error in modcov() */
   Real_Vector temp, temp2,             /* Temp storage */
     ref,                                                           /* BPSK modulated signal */
     chips,                                                 /* PN Sequence */
     coded_data,                            /* Sampled & PN-modulated data signal */
```

```
       interferer,                                   /* Interferer */
       noise,                                                        /* Total noise environment */
       rcvd_tr,                                              /* Received signal (r=s+n) */
       whitened,                                             /* Whitened rcvd_tr */
       a_hat,                                                        /* AR parameter estimate vector */
       a_hat_neg1,                                   /* As above, except a_hat_neg1[0]=-1, length = 1 */
       g_lo, g_ar, g_lw,                     /* Initialize LO, AR, and LO on Wh. detectors */
       g_temp,                                                       /* Temp storage for g, holding one symbol's worth */
       lc_dec,                                                       /* linear correlator decision vector */
       lo_dec,                                                       /* Locally Optimium decision vector */
       lw_dec,                                                       /* LO on whitened decision vector */
       ar_dec,                                                       /* ARLO decision vector */
       wc_dec,                                                       /* lc on whitened decision vector */
       Pb_lc_avg,                                            /* Average P[err] */
       Pb_wc_avg, Pb_lo_avg, Pb_lw_avg, Pb_ar_avg, block,  /* Block to prepare for histogram */
       symbol,                                                       /* One symbol */
       symbolindx,                                   /* Index to separate out symbols from bitstream */
       filsym,                                                       /* whitened symbol */
       h_lo, h_wh,                                   /* LO h() & AR h() */
       bw_lo, bw_wh,                                         /* LO & AR bin_width */
       min_lo, min_wh;                               /* min value of histogram/pdf LO & AR */
     Real_Matrix Pb_lc,                              /* P[err] for lc */
       Pb_wc, Pb_lo, Pb_lw, Pb_ar;
     FILE *Pblc, *Pbwc, *Pblo, *Pblw, *Pbar, *snr_vals,
            *interferer_out, *noise_out, *rcvd_tr_out, *a_hat_out, *ref_out;
     long ltime;                                             /* For rand() usage */
     if (argc != 2)                                          /* Parameter file check */
            {
                   printf ("You forgot to enter the parameter filename.\n");
                   exit (1);
            }
     read_parms (argv[1]);                   /* Read simulation parameters from file */
/********* Vector and matrix creation **********/
     ref = valloc (NULL, B);                 /* Create signal */
     coded_data = valloc (NULL, Nb * B);   /* Sampled & PN-modulated data signal */
     chips = valloc (NULL, Nb * B);        /* Chip sequence */
     interferer = valloc (NULL, Ntot);     /* Create interferer */
     noise = valloc (NULL, Ntot);
     rcvd_tr = valloc (NULL, Ntot);
     whitened = valloc (NULL, Ntot);       /* whitened rcvd_tr */
     a_hat = valloc (NULL, P + 1); /* AR parameter estimate vector */
     a_hat_neg1 = valloc (NULL, 1);            /* a_hat[0]=-1, length(a_hat) = 1 */
     g_lo = valloc (NULL, Ntot);     /* LO detector */
     g_ar = valloc (NULL, Ntot);     /* AR detector */
     g_lw = valloc (NULL, Ntot);     /* LO on Wh. detector */
     g_temp = valloc (NULL, Nb);     /* To hold one symbol's worth of information */
     Pb_lc = mxalloc (NULL, it, Eb_max - Eb_min + 1);
     Pb_wc = mxalloc (NULL, it, Eb_max - Eb_min + 1);
     Pb_lo = mxalloc (NULL, it, Eb_max - Eb_min + 1);
     Pb_lw = mxalloc (NULL, it, Eb_max - Eb_min + 1);
     Pb_ar = mxalloc (NULL, it, Eb_max - Eb_min + 1);
     Pb_lc_avg = valloc (NULL, Eb_max - Eb_min + 1);
     Pb_wc_avg = valloc (NULL, Eb_max - Eb_min + 1);
     Pb_lo_avg = valloc (NULL, Eb_max - Eb_min + 1);
     Pb_lw_avg = valloc (NULL, Eb_max - Eb_min + 1);
     Pb_ar_avg = valloc (NULL, Eb_max - Eb_min + 1);
     block = valloc (NULL, Bh * Nb);       /* block to prepare for histogram */
     symbol = valloc (NULL, Nb);     /* Single symbol */
     symbolindx = valloc (NULL, Nb);
     filsym = valloc (NULL, Nb);     /* whitened symbol */
     wc_dec = valloc (NULL, B);      /* allocate decision vector */
     lc_dec = valloc (NULL, B);
     lo_dec = valloc (NULL, B);
     lw_dec = valloc (NULL, B);
     ar_dec = valloc (NULL, B);
     h_lo = valloc (NULL, K + 1);  /* "h" function */
     h_wh = valloc (NULL, K + 1);
     bw_lo = valloc (NULL, 1);       /* bin_width LO & AR */
     bw_wh = valloc (NULL, 1);
     min_lo = valloc (NULL, 1);      /* min value of histogram LO & AR */
     min_wh = valloc (NULL, 1);
/*********** End creation section ***********/
     a_hat_neg1[0] = -1.0;                   /* Assign value */
     saveData = 1;                                   /* Save data first time through */
     where_running = 1;                              /* 1=Sundevil, 2=L90 */
     for (snr = 0; snr < Eb_max - Eb_min + 1; snr++)      /* Iterate over each snr */
            {
                   for (pe_avg = 0; pe_avg < it; pe_avg++) /* Iterate Monte Carlo */
                     {
                            /* For rand() */
                            ltime = time (NULL);
                            srand ((unsigned) ltime / 2);  /* compute seed for rand() */
                            for (i = 0; i < B; i++)          /* Create random signal */
                                   {
                                     *(ref + i) = (2 * (Real) rand () / RAND_MAX) - 1.0;
                                     if (*(ref + i) < 0)
                                            *(ref + i) = -1.0;
                                     else if (*(ref + i) > 0)
                                            *(ref + i) = 1.0;
                                     else
                                            *(ref + i) = 0.0;
```

```
        }
 sym_amp = sqrt ((double) sigmat2 * Eb_sigma[snr]);     /* Symbol amplitude assuming chip-

 /* NOTE: Let the signal have amplitude of '1' and then divide the thermal noise
    by the symbol amplitude (below, in 'implement the channel' ) */
 amp = 1;
 dsss (ref, amp, coded_data, chips);    /* Transmitted DSSS signal */
 printf ("DSSS done, [it,snr]=[%i,%i]\n", pe_avg, snr); /* Progress indicator */
/*********** Implement the Channel **************/
 vinit (interferer, Ntot, 0.0); /* initialize interferer to zero */
 if ((Itype[0] == 'c') && (Itype[1] == 'w'))
        contwave (interferer);
 else if ((Itype[0] == 'p') && (Itype[1] == 'b'))
        th_inter (interferer);
 else if ((Itype[0] == 'm') && (Itype[1] == 'x'))
        {
          th_inter (interferer);
          contwave (interferer);
        }
 printf ("Interferer done, [it,snr]=[%i,%i]\n", pe_avg, snr);   /* Progress Indicator */
/********** Add thermal noise with a Gaussian pdf of N(mu=0, sigmat^2)
to the correlated interferer (and additive channel is assumed ) */
 for (i = 0; i < Ntot; i++)
        *(noise + i) = *(interferer + i) +
          (((double) sigmat / sym_amp) * normal (1, 0));
/*********** Implement the observed signal ****************/
 receiver (coded_data, noise, limit, rcvd_tr);
 printf ("Receiver done [it,snr]=[%i,%i]\n", pe_avg, snr);      /* Progress Indicator */
/*********** PRINT OUT data for spectrum plots *************/
 if (saveData == 1)
        {
          // For Sundevil
          if (where_running == 1)
                {
                        interferer_out =
                          fopen
                          ("x:\\documents\\research\\conversion\\results\\interferer_out.txt",
                                "w");
                        noise_out =
                          fopen
                          ("x:\\documents\\research\\conversion\\results\\noise_out.txt",
                                "w");
                        rcvd_tr_out =
                          fopen
                          ("x:\\documents\\research\\conversion\\results\\rcvd_tr_out.txt",
                                "w");
                        a_hat_out =
                          fopen
                          ("x:\\documents\\research\\conversion\\results\\a_hat_out.txt",
                                "w");
                }
          // For L90
          if (where_running == 2)
                {
                        interferer_out =
                          fopen ("c:\\temp\\mike\\results\\interferer_out.txt",
                                                 "w");
                        noise_out =
                          fopen ("c:\\temp\\mike\\results\\noise_out.txt", "w");
                        rcvd_tr_out =
                          fopen ("c:\\temp\\mike\\results\\rcvd_tr_out.txt", "w");
                        a_hat_out =
                          fopen ("c:\\temp\\mike\\results\\a_hat_out.txt", "w");
                }
          for (i = 0; i < Ntot; i++)
                {
                        fprintf (interferer_out, "%f\n", interferer[i]);
                        fprintf (noise_out, "%f\n", noise[i]);
                        fprintf (rcvd_tr_out, "%f\n", rcvd_tr[i]);
                }
          fclose (interferer_out);
          fclose (noise_out);
          fclose (rcvd_tr_out);
          saveData = 0;   // Don't save data anymore
        }
/************ Implement a linear receiver with truncation **************/
 decision (chips, rcvd_tr, lc_dec);
 printf ("LC: Decision done, [it,snr]=[%i,%i]\n", pe_avg, snr); /* Progress Indicator */
 for (i = 0, num_errs_lc = 0.0; i < B; i++)     /* Find no. of error for linear correlator */
        num_errs_lc = num_errs_lc + (0.5 * abs ((lc_dec[i] - ref[i])));
/************* Implement LO Detector Techniques *********************/
 /* Find the appropriate whitener */
 j = (int) (floor (Ntot / (B / Bar)) - 1);      /* last array element of interest */
 temp = valloc (NULL, j);       /* allocate for passing in partial rcvd_tr */
 for (i = 0; i < j; i++)
        *(temp + i) = *(rcvd_tr + i);   /* Copy rcvd_tr[] into temp[] */
 modcov (temp, j, P, a_hat, error);     /* Find the whitener */
 vfree (temp);                  /* clean up */
/*********** Implement a linear correlator on the whitened signal *******/
 temp = valloc (NULL, 1);       /* create vector with one element = 1 */
 temp[0] = 1;
```

```
                          /* find whitened received signal */
                          filter (vscale (a_hat, P + 1, -1.0), P + 1, temp, 1, rcvd_tr,
                                                  Ntot, whitened);
                          vfree (temp);                   /* clean up */
                          decision (chips, whitened, wc_dec);    /* make decisions */
                          printf ("LC on Wh: Decision done, [it,snr]=[%i,%i]\n", pe_avg, snr);   /* Progress */
                          for (i = 0, num_errs_wc = 0.0; i < B; i++)     /* Find no. of errors for LC on wh. */
                                 num_errs_wc = num_errs_wc + (0.5 * abs ((wc_dec[i] - ref[i])));
                         /************** Implement ARLO and LO Receivers ***************/
                          /* Preparation */
                          vinit (g_lo, Ntot, 0.0);       /* Initialize to 0.0 */
                          vinit (g_ar, Ntot, 0.0);
                          vinit (g_lw, Ntot, 0.0);
                          Nsamh = Bh * Nb;                  /* Number of samples in each histogram */
                          temp2 = valloc (NULL, 1);      /* Allocate for a_hat=-1 (P=0) */
                          temp2[0] = -1;
                          for (histonum = 0; histonum < ((int) floor (B / Bh)); histonum++)
                                 {                                                 /* Process Bh*Nb samples
                                                                                      at a time for a histogram */
                                   vinit (block, Nsamh, 0.0);    /* Init block to prepare for histogram */
                                   for (i = histonum * Nsamh, j = 0;
                                                  i < ((histonum + 1) * Nsamh - 1); i++, j++)
                                          *(block + j) = *(rcvd_tr + i); /* Apply the correct symbols */
                                   temp = valloc (NULL, 1);      /* Allocate temp[0] = -1 for LO case */
                                   temp[0] = -1;
                                   ar_histo (block, Bh * Nb, temp, 1, h_lo, bw_lo, min_lo);      /* LO (P=0) a_0=-1 */
                                   ar_histo (block, Bh * Nb, a_hat, P + 1, h_wh, bw_wh, min_wh); /* AR methods */
                                   vfree (temp);   /* clean up */
                                   for (symcnt = 0; symcnt < Bh; symcnt++)       /* Apply nonlinearity to each symbol */
                                          {
                                                 vinit (symbol, Nb, 0.0);        /* Look at one symbol */
                                                 i = histonum * Nsamh + ((symcnt - 1) * Nb);     /*Find indices of current symbol */
                                                 for (j = 0; j < Nb; j++)
                                                   {
                                                          *(symbolindx + j) = i + j;
                                                          *(symbol + j) = *(rcvd_tr + i + j);    /* Get current symbol */
                                                          *(filsym + j) = *(whitened + i + j);   /* Get current whitened symbol */
                                                   }
                                                 temp = valloc (NULL, Nb);       /* To hold g_xx[symbolindx] */
                                                 temp = nlp (symbol, Nb, min_lo, h_lo, K + 1, bw_lo, a_hat_neg1, 1, Nb); /* lo detector */
                                                 for (i = symbolindx[0], j = 0; j < Nb; i++, j++)        /* Assign g_lo = temp */
                                                   *(g_lo + i) = *(temp + j);
                                                 temp = nlp (filsym, Nb, min_wh, h_wh, K + 1, bw_wh, a_hat_neg1, 1, Nb); /* lw detector */
                                                 for (i = symbolindx[0], j = 0; j < Nb; i++, j++)        /* Assign g_lw = temp */
                                                   *(g_lw + i) = *(temp + j);
                                                 temp = nlp (symbol, Nb, min_wh, h_wh, K + 1, bw_wh, a_hat, P + 1, Nb);  /* ar detector */
                                                 for (i = symbolindx[0], j = 0; j < Nb; i++, j++)        /* Assign g_ar = temp */
                                                   *(g_lw + i) = *(temp + j);
                                          }
                                 }
                          vfree (temp2);
                         /*************** Make ARLO and LO Detector Decisions ***********************/
                          decision (chips, g_lo, lo_dec);        /* make decisions */
                          printf ("LO: Decision done, [it,snr]=[%i,%i]\n", pe_avg, snr); /* Progress */
                          decision (chips, g_lw, lw_dec);
                          printf ("LW: Decision done, [it,snr]=[%i,%i]\n", pe_avg, snr); /* Progress */
                          decision (chips, g_ar, ar_dec);
                          printf ("AR: Decision done, [it,snr]=[%i,%i]\n", pe_avg, snr); /* Progress */
                          for (i = 0, num_errs_lo = 0.0, num_errs_lw = 0.0, num_errs_ar = 0.0; i < B; i++)       /* Count number of errors */
                                 {
                                   num_errs_lo = num_errs_lo + (0.5 * abs ((lo_dec[i] - ref[i])));       /* lo errors */
                                   num_errs_lw = num_errs_lw + (0.5 * abs ((lw_dec[i] - ref[i])));       /* lw errors */
                                   num_errs_ar = num_errs_ar + (0.5 * abs ((ar_dec[i] - ref[i])));       /* ar errors */
                                 }
                          //vfree(lo_dec);vfree(lw_dec);vfree(ar_dec);
                         /************* Compute probability of bit error for this MC iteration and this snr */
                          Pb_lc[pe_avg][snr] = num_errs_lc / B;  /* linear correlator */
                          Pb_wc[pe_avg][snr] = num_errs_wc / B;
                          Pb_lo[pe_avg][snr] = num_errs_lo / B;
                          Pb_lw[pe_avg][snr] = num_errs_lw / B;
                          Pb_ar[pe_avg][snr] = num_errs_ar / B;
                          printf ("Monte Carlo Iteration %i done, [snr]=[%i]\n\n", pe_avg, snr); /* Progress Indicator */
                   }                                                        /* End Monte Carlo Iteration */
                 printf ("SNR iteration %i done. \n\n", Eb_min + snr);   /* Progress Indicator */
          }                                                                         /* End SNR iteration */
/************** Compute average P[err] from MC analysis ***********/
   temp = valloc (NULL, 5);               /* 5 different receivers */
   for (i = 0; i < (Eb_max - Eb_min + 1); i++)
          {
                 vinit (temp, 5, 0.0);     /* Initialize temp to zeros */
                 for (j = 0; j < it; j++)  /* Sum up over all MC iterations */
                   {
                          temp[0] = temp[0] + Pb_lc[j][i];       /* linear correlator */
                          temp[1] = temp[1] + Pb_wc[j][i];
                          temp[2] = temp[2] + Pb_lo[j][i];
                          temp[3] = temp[3] + Pb_lw[j][i];
                          temp[4] = temp[4] + Pb_ar[j][i];
                   }
                 useinput_ = 1;                          /* Perform in-place */
                 temp = vscale (temp, 5, 1.0 / it);      /* Average over MC */
                 useinput_ = 0;                          /* Reset */
```

```
              Pb_lc_avg[i] = temp[0];
              Pb_wc_avg[i] = temp[1];
              Pb_lo_avg[i] = temp[2];
              Pb_lw_avg[i] = temp[3];
              Pb_ar_avg[i] = temp[4];
        }
   vfree (temp);                                    /* Clean up */
        /************ Save to disk P[err] info *************/
   // For sundevil
   if (where_running == 1)
        {
              snr_vals =
                fopen ("x:\\documents\\research\\conversion\\results\\snr_vals_.txt",
                                        "w");
              Pblc =
                fopen ("x:\\documents\\research\\conversion\\results\\pblc.txt", "w");
              Pbwc =
                fopen ("x:\\documents\\research\\conversion\\results\\pbwc.txt", "w");
              Pblo =
                fopen ("x:\\documents\\research\\conversion\\results\\pblo.txt", "w");
              Pblw =
                fopen ("x:\\documents\\research\\conversion\\results\\pblw.txt", "w");
              Pbar =
                fopen ("x:\\documents\\research\\conversion\\results\\pbar.txt", "w");
        }
   // For L90
   if (where_running == 2)
        {
              snr_vals = fopen ("c:\\temp\\mike\\results\\snr_vals_.txt", "w");
              Pblc = fopen ("c:\\temp\\mike\\results\\pblc.txt", "w");
              Pbwc = fopen ("c:\\temp\\mike\\results\\pbwc.txt", "w");
              Pblo = fopen ("c:\\temp\\mike\\results\\pblo.txt", "w");
              Pblw = fopen ("c:\\temp\\mike\\results\\pblw.txt", "w");
              Pbar = fopen ("c:\\temp\\mike\\results\\pbar.txt", "w");
        }
   for (i = 0; i < (Eb_max - Eb_min + 1); i++)
        {
              fprintf (snr_vals, "%i\n", Eb_min + i); /* Save snr vector */
              fprintf (Pblc, "%f\n", Pb_lc_avg[i]);   /* linear correlator */
              fprintf (Pbwc, "%f\n", Pb_wc_avg[i]);
              fprintf (Pblo, "%f\n", Pb_lo_avg[i]);
              fprintf (Pblw, "%f\n", Pb_lw_avg[i]);
              fprintf (Pbar, "%f\n", Pb_ar_avg[i]);
        }
   /* Clean house */
   fclose (Pblc);
   fclose (Pbwc);
   fclose (Pblo);
   fclose (Pblw);
   fclose (Pbar);
   fclose (snr_vals);
   mathfree ();
}
```

A.2.2 AR_Histo.c

```
/* AR_histo.c
  *             by Mike Banys
  *             March 3, 2001
  *
  * This estimates the pdf of a white noise sequence that drives a P-th
  * order auto-regressive filter model. It returns the vector function
  * h() which is not the pdf, but rather the natural log of the
  * derivative of the pdf, to be used to the LO non-linear processor.
  *
  *
  */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "mathlib.h"
#include "sim_parameters.h"
#include "sim_parameters_extern.h"
voidar_histo (Real_Vector stimulus, unsigned stimulus_length, Real_Vector a,
                                   unsigned a_length,
                                   Real_Vector h, Real_Vector bin_width, Real_Vector mini)
                                 /* bin_width & mini are vectors to enable pass-by-reference */
{
   int i, j;                                                   /* counter variable */
   unsigned N;                                            /* samples for histogram */
   Real_Vector white, filter_a_variable, /* used for sending -1 into filter() */
     pdf,                                                             /* pdf of white noise (frequency count of histogram) */
     bp,                                                       /* histogram bin centers */
     neg_a;                                                    /* Temporary variable to hold -a */
   white = valloc (NULL, stimulus_length);
   if (a_length == 1)
         vcopy (white, stimulus, stimulus_length);      /* using ARLO */
   else
```

```
          /* using filtered LO */
          {
                 filter_a_variable = valloc (NULL, 1);
                 filter_a_variable[0] = 1;
                 neg_a = vscale (a, a_length, -1.0);
                 filter (neg_a, a_length, filter_a_variable, 1, stimulus,
                                  stimulus_length, white);
                 vfree (neg_a);
          }
/* Build the pdf estimate via the histogram with K bins */
   N = Nb * Bh;                                        /* samples calculated for this histogram */
   pdf = valloc (NULL, K);
   bp = valloc (NULL, K);
   hist (white, stimulus_length, K, pdf, bp);    /* Compute histogram estimate of white noise */
   for (i = 0; i < K; i++)                 /* Find possible negative values or zeros */
          {
                 if (*(pdf + i) <= 0.0)    /* set them equal to 1.0 for calculation of g */
                   *(pdf + i) = 1.0;
          }
   bin_width[0] = (Real) * (bp + 1) - *bp;       /* Width of each equi-spaced bin, typecasted to Real */
   mini[0] = vminval (white, stimulus_length, &i);       /* Minimum value in the histogram */
   useinput_ = 1;                                      /* overwrite pdf[] with scaled pdf[] */
   pdf = vscale (pdf, K, (1 / (*bin_width * N)));        /* scale histogram so that pdf integrates to 1 */
   useinput_ = 0;                                      /* reset */
/* The "h" function is the derivative of the ln of the pdf of the white noise sequence */
   *h = (1.0 / *bin_width) * (log (*(pdf + 1)) - log (*pdf));
   for (i = 1; i < K - 1; i++)
          *(h + i) =
                 (0.5 / *bin_width) * (log (*(pdf + i + 1)) - log (*(pdf + i - 1)));
   *(h + K - 1) =
          (1.0 / *bin_width) * (log (*(pdf + K - 1)) - log (*(pdf + K - 2)));
   *(h + K) = 0.0;                                     /* For received values outside support of h (a junkbin) */
/* Clean House */
   vfree (filter_a_variable);
   vfree (pdf);
   vfree (bp);
   vfree (white);
}
```

A.2.3 AR_Theory.c

```
/*AR Theory
  *      by Fernando Martinez Vallina
  *
  * Calculates the pdf of a white noise sequence that drives the P-th
  * auto-regressive filter model. Returns a function h() that is the
  * natural log of the derivative of the pdf.
  *
  *
  * Inputs:      K        - Number of data points
  *              T        - Support of the pdf
  *              iidtype  - Type of the white noise pdf
  *              iidparms - Parameters of the pdf
  *              ISR      - Interferer to Signal Ratio (dB)
  *
  * Outputs:     h        - d/d(rho) of ln(fw())
  *
  *
  * This is based on the original AR Theory code in MATLAB
  */
#include <math.h>
#include <stdlib.h>
#include "mathlib.h"
#include "sim_parameters.h"
#include "sim_parameters_extern.h"
void
ar_theory (int iidparms[], char iidtype[], int ISR, Real K,
                          Real T, Real_Vector h)
{
   Real xstep, xmax, sig1, power;
   int length, i;                                      //length is the length of the sequence, i is a counting variable
   Real_Vector x, mu, sig;
   power = pow (10, (ISR / 10));
   xmax = T / 2;
   xstep = T / (K - 1);
   length = T / xstep;
   x = valloc (NULL, length);
   x[0] = -xmax;
   for (i = 1; i <= length; i++)
          {
                 x[i] = x[i - 1] + xstep;
          }
   h = valloc (NULL, length);
   if (iidtype == 'gs')
          {
                 mu = valloc (NULL, length);
                 for (i = 0; i < length; i++)
                   {
                        mu[i] = iidparms[1];
                  }
                sig = valloc (NULL, length);
                for (i = 0; i < length; i++)
                  {
                        sig[i] = iidparms[2];
                  }
                for (i = 0; i < length; i++)
                  {
                        h[i] = -(x[i] - mu[i]) / (power * sig[i] * sig[i]);
                  }
        }
   if (iidtype == 'lp')
        {
                sig1 = iidparms[1];
                sig1 = sig1 * sig1 * power;
        }
   vfree (x);
   vfree (mu);
   vfree (sig);
}
```

A.2.4 Contwave.c

```
/* interferer must be initialized to zero (or contain values from
another interference type) before calling this function
*/

#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include "mathlib.h"
#include "sim_parameters_extern.h"
   void
contwave (Real_Vector interferer)
{
   Real amp, /* Individual interferer amplitude */
        delta_t;                                                /* time interval */
   int i, intf;                                 /* counter */
   Real_Vector time, /* Time sequence */
        phase;                                                  /* phase of each interferer */
   double pi = 3.1415926535;
   amp = sqrt (2 * pow (10, ISR / 10) / I);     /* Individual interferer amplitudes */
   phase = valloc (NULL, I);
   for (i = 0; i < I; i++)                /* Set each interferer's phase to 0 */
        *(phase + i) = 0;
   delta_t = (Real) 1.0 / Ntot;  /* Time interval */
   time = valloc (NULL, Ntot); /* Time sequence */
   for (i = 0; i < Ntot; i++)
        *(time + i) = delta_t * i;
   for (intf = 0; intf < I; intf++)     /* Create sinusoid interferer */

        {
                for (i = 0; i < Ntot; i++)
                  *(interferer + i) =
                        *(interferer + i) +
                        (amp * sin (2 * pi * freq[intf] * time[i] + phase[intf]));
        }
   vfree (time);
   vfree (phase);
}
```

A.2.5 Conv.c

```
/* Convolution Function
  *
  *             by Michael Banys
  *
  * conv(v1, length1, v2, length2, output)
  *
  *   Inputs:    v1 = Vector 1
  *                             v2 = Vector 2
  *                             length1 = length(v1)
  *                             length2 = length(v2)
  *
  *      Outputs:        output = convolution of vec1 with vec2
  *
  * "The resulting vector is output_length=length1+length2-1.
  *  If v1 and v2 are vectors of polynomial coefficients, convolving
  *  them is equivalent to multiplying the two polynomials."
  *
  * This is based on MATLAB's conv function.
  */
#include <stdlib.h>
#include "mathlib.h"
```

```
#include "sim_parameters_extern.h"
voidconv (Real_Vector v1, int length1, Real_Vector v2,
                          int length2, Real_Vector output)
{
   /* Use same format as filter() for this operation.
    * conv(a,b) is the same as conv(b,a), but we can make it
    * go substantially faster if we swap arguments to make the first
    * argument the shorter of the two.
    */
   int i,                                                                      /* Counter variable */
     filter_order,                                             /* Order of numerator */
     filter_iteration,                     /* Variable tracking loop through input data */
     element_num,                                              /* Variable indicating current coefficient and delay
                                                                                         being processed */
     first_length,                                             /* Length of vector that is first into conv */
     second_length;                                    /* Length of vector that is used as data_in */
   Real_Vector v_alt,                                  /* Temp vector for zero-padding v1 or v2 */
    *first_v,                                                  /* Pointer for first argument into convolution,
                                                                                         as explained below.  May be assigned to v1 or v2 */
    *second_v,                                                 /* as above */
     filter_delay;                                             /* Filter delay values */
   Real sumx,                                                  /* Temporary variable representing sum of elements of a[]
                                                                multiplied by elements of filter_delay[]  */
     sumy;                                                    /* Temporary variable repereseting sum of elements of b[]
                                                               multiplied by elements of filter_delay[]   */
   if (length1 > length2)
          {
                 if (length2 > 1)
                   {
                          v_alt = valloc (NULL, (length1 + length2 - 1));        /* For zero-padding v1 */
                          for (i = 0; i < length1; i++)
                                 *(v_alt + i) = *(v1 + i);       /* Copy v1 into v_alt */
                          for (i; i < (length1 + length2 - 1); i++)      /* Zero pad rest of v_alt */
                                 *(v_alt + i) = 0;
                   }
                 first_v = &v2;                                /* v2 will be first vector into the conv operation */
                 second_v = &v_alt;
                 first_length = length2;
                 second_length = length1;
          }
   else
          {
                 if (length1 > 1)
                   {
                          v_alt = valloc (NULL, (length1 + length2 - 1));        /* For zero-padding v2 */
                          for (i = 0; i < length2; i++)
                                 *(v_alt + i) = *(v2 + i);       /* Copy v2 into v_alt */
                          for (i; i < (length1 + length2 - 1); i++)
                                 *(v_alt + i) = 0;         /* Zero pad rest of v_alt */
                   }
                 first_v = &v1;                                /* v1 will be first vector into the conv operation */
                 second_v = &v_alt;
                 first_length = length1;
                 second_length = length2;
          }
   /* Begin actual convolution using Direct Form II structure, as in filter() */
   filter_delay = valloc (NULL, first_length);   /* Allocate memory for filter_delay length */
   filter_order = first_length;  /* Set order of numerator */
   for (i = 0; i < first_length; i++)    /* Set w[n] delay array, filter_delay, to all zeros */
          *(filter_delay + i) = 0;
   for (filter_iteration = 0; filter_iteration < (first_length + second_length - 1); filter_iteration++) /* Start Filtering */
          {
                 sumx = sumy = 0.0;                /* Initialize variables */
                 /* Can SKIP summing left side, as there are no denominator coefficients */
                 /* Sum right side of Direct Form II Structure */
                 for (element_num = 1; element_num < first_length; element_num++)
                   sumy = sumy + (*(*first_v + element_num) * filter_delay[element_num]);
                 /* Compute current filter delay (filter_delay[0]) */
                 *filter_delay = *(*second_v + filter_iteration) + sumx;
                 /* Compute output element, y(n) (y[filter_iteration]) */
                 output[filter_iteration] = *(*first_v) * filter_delay[0] + sumy;
                 /* Delay each filter_delay[] element by one */
                 for (element_num = first_length - 1; element_num > 0; element_num--)
                   filter_delay[element_num] = filter_delay[element_num - 1];
          }
   vfree (filter_delay);
   vfree (v_alt);
}
```

A.2.6 Decision.c

```
#include <stdlib.h>
#include <math.h>
#include "mathlib.h"
#include "sim_parameters_extern.h"
   void
decision (Real_Vector pn, Real_Vector g, Real_Vector dec)
{
   int i, j, k;                                    /* Counter */
   Real corr;                                              /* Correlation of pn sequence with g */
   for (i = 0, k = 0; i < Ntot; i = i + Nb, k++)

          {
                 for (j = 0, corr = 0.0; j < Nb; j++)    /* Correlate each symbol with pn and SUM each correlation */
                   corr = corr + (pn[i + j] * g[i + j]);
                 if (corr <= 0)
                   *(dec + k) = -1;

                 else
                   *(dec + k) = 1;
          }
}
```

A.2.7 DSSS.c

```
#include <time.h>
#include <stdlib.h>
#include "mathlib.h"
#include "sim_parameters_extern.h"
voiddsss (Real_Vector ref, int amp, Real_Vector coded_data, Real_Vector chips)
{
   unsigned int i,                                         /* temp var */
     j,                                                            /* temp var */
     k;
   Real_Vector data, tmp;
   long ltime;                                             /* For random seed */
   data = valloc (NULL, Ntot);     /* Sample each symbol Nb times (Nb samples per symbol) */
   for (i = 0, k = 0; i < Ntot; i = i + Nb, k++) /* compute data vector */
          {
                 for (j = 0; j < Nb; j++)
                   *(data + i + j) = *(ref + k);
          }
   tmp = valloc (NULL, B * PG);
   ltime = time (NULL);
   srand ((unsigned) ltime / 2);
   for (i = 0; i < B * PG; i++)  /* Create PN chipping sequence */
          {
                 *(tmp + i) = (2 * (Real) rand () / RAND_MAX) - 1.0;     /* PN sequence B*PG chips long */
                 if (*(tmp + i) < 0)               /* Random chips at 1 sample per chip */
                   *(tmp + i) = -1.0;
                 else if (*(tmp + i) > 0)
                   *(tmp + i) = 1.0;
                 else
                   *(tmp + i) = 0.0;
          }
   for (i = 0, k = 0; i < Ntot; i = i + Nc, k++) /* Sample each chip Nc times */
          {
                 for (j = 0; j < Nc; j++)
                   *(chips + i + j) = *(tmp + k);
          }
   for (i = 0; i < Ntot; i++)      /* The sampled signal modulated by the PN sequence */
          *(coded_data + i) = amp * (*(chips + i)) * (*(data + i));
   vfree (data);
   vfree (tmp);
}
```

A.2.8 FFTShift.c

```
/* FFTSHIFT.c
  *
  * Swaps the left and right halves of the input vector, placing the DC
  * component in the middle of the spectrum.
  * i.e. a = [0 1 2 3 4] -->             [3 4 0 1 2]
  * i.e. a = [0 1 2 3 4 5] -->   [ 3 4 5 0 1 2]
  *
  * v1 = input complex vector
  * length1 = length of complex vector
  *
  * by Michael Banys
  * February 7, 2000
  */
#include <stdlib.h>
#include <math.h>
#include "mathlib.h"
#include "sim_parameters_extern.h"
voidfftshift (Complex_Vector v1, unsigned int length1)
{
   unsigned int k,                                         /* counter */
     j, midpoint;                                          /* Middle of v1 */
   Complex temp1,                                          /* temp Storage */
     temp2;
   midpoint = (unsigned int) ceil ((double) length1 / 2) - 1;    /* Find midpoint of v1 */
   /* if length1 is odd, this finds exact midpoint.
```

```
     * if length1 is even, this find the lesser of the two midpoints */
    if (fmod ((double) length1, 2.0))     /* If v1 contains ODD number of elements */
           {
                   temp2.r = v1[midpoint].r;
                   temp2.i = v1[midpoint].i;
                   for (k = midpoint, j = (length1 - 1); k >= 1; k--, j--) /* Iterate from midpoint to zero */
                     {
                           temp1.r = v1[j].r;        /* Save first storage location */
                           temp1.i = v1[j].i;
                           v1[j].r = temp2.r;        /* Move a value */
                           v1[j].i = temp2.i;
                           temp2.r = v1[k - 1].r; /* Save second storage location */
                           temp2.i = v1[k - 1].i;
                           v1[k - 1].r = temp1.r; /* Move a value */
                           v1[k - 1].i = temp1.i;
                     }
                   v1[midpoint].r = temp2.r;
                   v1[midpoint].i = temp2.i;
           }
    else
           /* If v1 contains EVEN number of elements */
           {
                   for (k = midpoint, j = (length1 - 1); k > 0; k--, j--)
                     {
                           temp1.r = v1[j].r;        /* Save value */
                           temp1.i = v1[j].i;
                           v1[j].r = v1[k].r;        /* Store value */
                           v1[j].i = v1[k].i;
                           v1[k].r = temp1.r;        /* Store value */
                           v1[k].i = temp1.i;
                     }
                   temp1.r = v1[j].r;                /* Save value */
                   temp1.i = v1[j].i;
                   v1[j].r = v1[k].r;                /* Store value */
                   v1[j].i = v1[k].i;
                   v1[k].r = temp1.r;                /* Store value */
                   v1[k].i = temp1.i;
           }
}
```

A.2.9 Filter.c

```
/* filter.c
  * Version 2.0, 5/1/01
  *      Written by Michael Banys & Angela Kaczmarski
  *
  * Revision Control:
  *      2.0 Changed to allow a denominator of 1
  *      1.0 Original version
  *
  */
/* OUTPUT = filter( b[M], M, a[N], N, data_in[input_length], input_length, y[] )
  *
  * b[M] = coefficients of numerator of impulse response
  * a[N] = coefficients of denominator of impulse response
  *      = order of filter
  *
  * This function computes the convolution
  * of vector "data_in" with the causal filter
  * whose time-domain filter coefficients
  * are of the following format:
  *
  *             a(1)*y(n) = -SUM(k=1,N) [ a(k)*y(n-k) ]
  *                                   +SUM(k=0,M) [ b(k)*x(n-k) ]
  *
  * If a(0) != 1, FILTER will normalize the filter
  * coefficients by a(0). If a(0)==0, return ERROR.
  *
  */
#include <stdlib.h>
#include "mathlib.h"
#define ORDER_ERROR     "Filter numerator order is greater than denominator order\n"
voidfilter (Real_Vector b, int M,
                                  Real_Vector a, int N,
                                  Real_Vector data_in, int input_length, Real_Vector y)
{
   int filter_order,                              /* Order of acting filter */
     filter_iteration = 0,                 /* Variable tracking loop through input data */
     element_num,                                        /* Variable indicating current coefficient and delay being
                                                                                  processed */
     i = 0;                                                      /* Iteration variable */
   Real normalization_value,       /* a[0] coefficient value */
     sumx,                                                       /* Temporary variable representing sum of elements of a[]
                                                                                  multiplied by elements of filter_delay[]                  */
     sumy;                                                       /* Temporary variable reperesenting sum of elements of b[]
                                                                                  multiplied by elements of filter_delay[]                  */
   Real_Vector filter_delay;       /* Filter delay values */
   if (N >= M)                                           // added
```

```
        {
                filter_delay = valloc (NULL, N);        /* Allocate memory for filter_delay length */
                filter_order = N;                         /* Set order of filter */
        }
   else                                                           // added
        {
                filter_delay = valloc (NULL, M);        /* Allocate memory for filter_delay length *///added
                filter_order = M;                         /* Set order of filter *//// added
        }
   /* Check to see if the length of b[] < length of a[]
      If length b[] >= length a[], output error message and exit function */
   if ((M >= N) & (N != 1))                // added (N!=1)
        printf (ORDER_ERROR);
   /* Check to see if denominator of IR is normalized (i.e. does a[0]=1
      If denominator is not normalized, normalize IR by a[0] */
   if (a[0] != 1)
        {
                normalization_value = a[0];
                /* Normalize denominator */
                for (i = 0; i < N; i++)
                  a[i] = a[i] / normalization_value;
                /* Normalize numerator */
                for (i = 0; i < M; i++)
                  b[i] = b[i] / normalization_value;
        }
   /* Set w[n] delay array, filter_delay[], to all zeroes */
   for (i = 0; i < filter_order; i++)    // changed N to filter_order
        filter_delay[i] = 0;
   /* Start filtering process */
   for (filter_iteration = 0; filter_iteration < input_length;
                 filter_iteration++)
        {
                sumx = sumy = 0.0;                /* Initialize variables */
                /* Sum left side of Direct Form II Structure */
                for (element_num = 1; element_num < N; element_num++)
                  sumx = sumx - a[element_num] * filter_delay[element_num];
                /* Sum right side of Direct Form II Structure */
                for (element_num = 1; element_num < M; element_num++)
                  sumy = sumy + b[element_num] * filter_delay[element_num];
                /* Compute current filter delay (filter_delay[0]) */
                filter_delay[0] = data_in[filter_iteration] + sumx;
                /* Compute output element, y(n) (y[filter_iteration]) */
                y[filter_iteration] = b[0] * filter_delay[0] + sumy;
                /* Delay each filter_delay[] element by one */
                for (element_num = filter_order - 1; element_num > 0; element_num--)
                  filter_delay[element_num] = filter_delay[element_num - 1];
        }
   vfree (filter_delay);
}
```

A.2.10 FlipLR.c

```
/***This is the C version of the Matlab Fliplr() function****
***written by Tayo Ihimoyan, 04/2000******/
#include "cmt.h"
#include "mathlib.h"
/*---------------------------------------------------------------------------*/
Real_Matrix
fliplr (Real_Matrix M, unsigned m, unsigned n)
{
/*
   -------------------------------------------------------------------------
      fliplr() returns a  m-by-n matrix M whose elements
         are taken from M(m-by-n) with columns flipped in the left-right direction,
that is, about a vertical axis .

        On exit, the output matrix M[][] is the value of this routine.
*/
/*---------------------------------------------------------------------------*/
   unsigned i, j, k;
   Real l;
   if (M == NULL)
        {
                matherr_ ("fliplr", E_NULLPTR);
                return NULL;
        }
   if (n % 2 == 0)
        k = n / 2;
   else
        k = (n - 1) / 2;
   l = 0.0;
   for (i = 0; i < k; i++)
        {
                for (j = 0; j < m; j++)
                  {
                        l = M[j][i];
                        M[j][i] = M[j][n - 1 - i];
                        M[j][n - 1 - i] = l;
                  }
          }
   return M;
}
```

A.2.11 FlipUD.c

```
/***This is the C version of the Matlab Flipud() function****
***written by Tayo Ihimoyan, 04/2000******/

#include "cmt.h"
#include "mathlib.h"
/*---------------------------------------------------------------------------*/
   Real_Matrix flipud (Real_Matrix M, unsigned m, unsigned n)
{

/*
   ---------------------------------------------------------------------------
      flipud() returns a  m-by-n matrix M whose elements
         are taken from M(m-by-n) with rows flipped in the up-down direction,
that is, about a horizontal axis .

        On exit, the output matrix M[][] is the value of this routine.
*/
/*---------------------------------------------------------------------------*/
   unsigned i, j, k;
   Real l;
   if (M == NULL)
          {
                 matherr_ ("flipud", E_NULLPTR);
                 return NULL;
          }
   if (m % 2 == 0)
          k = m / 2;

   else
          k = (m - 1) / 2;
   l = 0;
   for (i = 0; i < k; i++)
          {
                 for (j = 0; j < n; j++)
                   {
                          l = M[i][j];
                          M[i][j] = M[m - 1 - i][j];
                          M[m - 1 - i][j] = l;
                   }
          }
   return M;
}
```

A.2.12 Hist.c

```
/* hist_mike.c
  * Version 1.0, 6/29/2001
  *      Written by Michael Banys
  * Revision Control:
  *
  *
  *
  *
  * hist( Real_Vector signal, int nbins, Real_Vector freq, Real_Vector bincenter )
  *
  *             freq            = vector containing frequency distribution of each bin
  *             bincenter       = vector containing center location of each bin
  *
  *             signal          = signal (vector) from which to compute histogram
  *             nbins           = number of bins necessary for histogram
  *
  *
  * The histogram function bins the elements of signal into the number of bins specified by
  * nbins.  The function returns the number of elements in each bin and the center point
  * of each bin.
  */
#include "mathlib.h"
#include <stdio.h>
#include <stdlib.h>
voidhist (Real_Vector signal, unsigned signal_length,
                          unsigned nbins, Real_Vector freq, Real_Vector bincenter)
{
   int j,                                                             /* Counter variables */
     binfound,                                          /* 0 if data not distributed into a bin; 1 if it is */
     currentbin;                                /* current bin being examined */
   Real min_signal, max_signal, binwidth;
   unsigned temp;
   min_signal = vminval (signal, signal_length, &temp);  /* Find min(signal) */
   max_signal = vmaxval (signal, signal_length, &temp);  /* Find max(signal) */
   binwidth = (max_signal - min_signal) / nbins; /* Calculate bin width */
   bincenter[0] = min_signal + 0.499999 * binwidth;      /* Center of first bin */
   /* use 0.499999 to properly sort min(signal) into first bin */
   for (j = 1; j < nbins; j++)
          {
                 *(bincenter + j) = *(bincenter + j - 1) + binwidth;     /* Find centers of rest of bins */
                 *(freq + j) = 0.0;                /* Initialize frequency vector to zero */
          }
   freq[0] = 0.0;                                          /* complete initialization of frequency vector to zero */
   for (j = 0; j < signal_length; j++)   /* Distribute signal[] into bins */
          {
                 binfound = 0;                             /* reset binfound? variable */
                 currentbin = 0;
                 if (*(signal + j) > (bincenter[nbins - 1] + 0.5 * binwidth))    /* if max(signal) falls outside of bin */
                   {
                          freq[nbins - 1] = freq[nbins - 1] + 1;
                          binfound = 1;
                   }
                 while (binfound == 0)
                   {
                          if ((*(signal + j) > (bincenter[currentbin] - 0.5 * binwidth)) &&
                                   (*(signal + j) <= (bincenter[currentbin] + 0.5 * binwidth)))
                                 {
                                   freq[currentbin] = freq[currentbin] + 1;      /* place signal data into current bin */
                                   binfound = 1;
                                 }
                          currentbin++;
                   }
          }
   freq[nbins - 1] = freq[nbins - 1] + 1;        /* Adjusted first bincenter causes max(signal)
                                                    not to be sorted into last bin. So add this
                                                    signal value into last bin manually. */
}
```

A.2.13 Modcov.c

```
/* Inputs:
  *      N = length(x)
  *      p = model order
  *      ARparm = AR parameter vector to return to caller
  *      error = return to caller
  */
#include "mathlib.h"
#include <stdlib.h>
#include "sim_parameters.h"
#include "sim_parameters_extern.h"
voidmodcov (Real_Vector x, unsigned N, unsigned p,
                                Real_Vector ARparm, Real error)
{
   int i, j;
   Real_Matrix X,                                          /* Toeplitz matrix of x */
     R,                                                            /* temp storage matrix for flipud/lr */
     R1, R2;
   Real_Vector b1,                                         /* temp storage vectors */
     b2, b_temp;
   b1 = valloc (NULL, (N - p));
   b2 = valloc (NULL, p + 1);
   for (i = p, j = p; i < N; i++, j--)   /* Copy vectors for use in Toeplitz */
          {
                 *(b1 + (i - p)) = x[i];
                 if (j >= 0)
                   *(b2 + (i - p)) = x[j];
          }
   X = toeplitz (b1, b2, (N - p), (p + 1));
   R = mxmul1 (X, X, p + 1, N - p, p + 1);       /* Generate the matrix of normal equations */
   vfree (b1);
   vfree (b2);                                             /* Clean house */
   R1 = mxalloc (NULL, p, p);      /* forward predictor-error matrix */
   for (i = 1; i < p + 1; i++)
          for (j = 1; j < p + 1; j++)
                 R1[i - 1][j - 1] = R[i][j];
   R2 = mxalloc (NULL, p, p);      /* Backward predictor-error matrix */
   for (i = 0; i < p; i++)
          for (j = 0; j < p; j++)
                 R2[i][j] = R[i][j];
   R2 = fliplr (R2, p, p);
   R2 = flipud (R2, p, p);
   b1 = valloc (NULL, p);                 /* Forware error coefficients */
   for (i = 1; i < p + 1; i++)
          *(b1 + i - 1) = R[i][0];
   b2 = valloc (NULL, p);                 /* Backward predictor-error coefficients */
   for (i = 0; i < p; i++)
          {
                 *(b2 + i) = R[p - (i + 1)][p];
          }
   ARparm[0] = -1;                                         /* AR Coefficients */
   b_temp = vmxmul (mxinv (mxadd (R1, R2, p, p), p), vadd (b1, b2, p), p, p);
   for (i = 1; i < p + 1; i++)
          ARparm[i] = *(b_temp + (i - 1));
   mxfree (R1);
   mxfree (R2);
   vfree (b_temp);
   vfree (b1);
   vfree (b2);                                              /* clean house */
   b1 = valloc (NULL, p + 1);      /* Begin Error calculation *//*R(1,:) */
   b2 = valloc (NULL, p + 1);      /* R(p+1,:) */
   for (i = 0; i < p + 1; i++)
          {
                 *(b1 + i) = R[0][i];              /* R(1,:) */
                 b2[p - i] = R[p][i];              /* fliplr( R(p+1,:) ) */
          }
   error = vdot (b1, ARparm, p + 1) + vdot (b2, ARparm, p + 1);
   vfree (b1);
   vfree (b2);
   mxfree (X);
   mxfree (R);                                              /* clean house */
}
```

A.2.14 NLP.c

```
#include <stdlib.h>
#include "mathlib.h"
#include <math.h>
#include "sim_parameters_extern.h"
Real_Vector
nlp (Real_Vector r, unsigned int r_length, Real_Vector mini, Real_Vector h,
          unsigned int h_length, Real_Vector bw, Real_Vector a,
          unsigned int a_length, unsigned int g_length)
{
   int i, j, k;
   Real_Matrix hargs,                              /* arguments for the "h" function */
     hindx,                                                /* for indexes of h function */
     tempmat;                                           /* temp matrix */
   Real_Vector rho, rho_1, rho_2,        /* temp usage for hargs */
     hargCol,                                           /* hargs column: temporary */
     g,                                                    /* vector to return to simulation.c */
     a_scaled;                                          /* Test */
   const Real inf = 30000000,      /* Represents inf in matlab */
     isinf = 100000;                              /* Min value for "essentially inf" response */
   Real num;                                                /* temporary storage */
   g = valloc (NULL, g_length);  /* allocate vector */
   rho = valloc (NULL, 2 * P + r_length);        /* Concatenate initial conditions & rho */
   for (i = 0; i < P; i++)
          {
                 rho[i] = 0.0;                              /* Initial Values of zero */
                 rho[i + P + r_length] = inf;    /* Fixes the indexing in the nonlinearity
                                                    (forces rhp(i>Nb) = Inf (which is outside
                                                    the support of the pdf and hence h() = 0!! */
          }
   for (i = P; i < P + r_length; i++)
          rho[i] = r[i - P];
   hargs = mxalloc (NULL, P + 1, Nb);    /* Initialize the args for "h" function */
   //tempmat = mxalloc(NULL, P+1, P+1); // allocated within Toeplitz.c
   rho_1 = valloc (NULL, P + 1);
   rho_2 = valloc (NULL, P + 1);
   a_scaled = valloc (NULL, a_length);
   a_scaled = vscale (a, a_length, -1.0);
   for (i = 0; i < Nb; i++)
          {
                 for (j = 0, k = i + P; j < P + 1; j++)  /* Allocate rho(indx+P:indx+2P) */
                   {                                                      /* and rho(indx+P:-1:indx) */
                         rho_1[j] = rho[i + P + j];
                         rho_2[j] = rho[k - j];
                   }
                 tempmat = toeplitz (rho_1, rho_2, P + 1, P + 1);        /* Get toeplitz */
                 hargCol = vmxmul (tempmat, a_scaled, P + 1, P + 1);
                 for (j = 0; j < P + 1; j++)     /* Copy hargcol into appropriate hargs column */
                   {
                         hargs[j][i] = hargCol[j];
                   }
                 mxfree (tempmat);
                 vfree (hargCol);
                 /* Now that we have the arguments (a matrix), we want to make use of the
                    histogram estimate of the pdf (actually the derivative of the natural log of
                    the pdf).  So we take the "hargs" matrix and change it to INDEXES of the
                    "h" function, utilizing the structure of the pdf (the lowest terms are on the
                    left side of the histogram).
                    This routine uses the trick of assuming the minimum values in the histogram
                    (mini) is located in the first bin, and then we use the combination of the
                    breakpoint information and the floor command to develop an integer that
                    puts the value of "hargs" in the correct bin. */
                 tempmat = mxalloc (NULL, P + 1, Nb);    /* Allocate matrix of ones */
                 mxinit (tempmat, P + 1, Nb, 1.0);
                 hindx = mxsub (hargs, mxscale (tempmat, P + 1, Nb, *mini), P + 1, Nb);
                 useinput_ = 1;                           /* overwrite input matrix */
```

```
       hindx = mxscale (hindx, P + 1, Nb, (1.0 / bw[0]));
       for (i = 0; i < P + 1; i++)     /* floor( hindx ) */
         for (j = 0; j < Nb; j++)
                hindx[i][j] = floor (hindx[i][j]);
       hindx = mxadd (hindx, tempmat, P + 1, Nb);
       useinput_ = 0;                            /* do not overwrite anymore (DEFAULT) */
       mxfree (hargs);                   // printf("1 ");
       mxfree (tempmat);                         // printf("2\n");
       /* Find arguments that lie outside the support of the "h" function and throw
          those values into a "junk" bin (assume their probability is approx. zero */
       for (i = 0; i < P + 1; i++)     /* Apply "junk" to junkbin */
         for (j = 0; j < Nb; j++)
                {
                       if ((hindx[i][j] < 1.0) || (hindx[i][j] > (h_length - 1)))
                         hindx[i][j] = (Real) h_length;
                }
       /* *** Develop non-linearity
          The following reshaping makes h(hindx) appear in the correct format
          We want P+1 rows of length Nb, where the arguments for "h" in each row
          are a linear combination of P+1 samples of the rho vector.  Each row is a
          shifted version to accomplish the necessary summations for g(rho) */
       tempmat = mxalloc (NULL, P + 1, Nb);    /* h(hindx) */
       for (i = 0; i < P + 1; i++)
         {
                for (j = 0; j < Nb; j++)
                       tempmat[i][j] = h[(int) (hindx[i][j] - 1)];
         }
       hargs = mxalloc (NULL, Nb, P + 1);      /* temp matrix for reshaping */
       hargs = mxtransp (tempmat, P + 1, Nb);  /* h(hindx') */
       mxfree (hindx);
       hindx = mxalloc (NULL, P + 1, Nb);
       hindx = mxtransp (hargs, Nb, P + 1);
       /* Perform a*reshape(h(hindx'),Nb,P+1)'; */
       for (j = 0; j < Nb; j++)  /* Multiply over each column */
         {
                num = 0.0;                                /* Initialize num to zero */
                for (i = 0; i < P + 1; i++)    /* Do each row by col multiplication */
                       num = num + (*(a + i) * hindx[i][j]);
                *(g + j) = num;
         }
       mxfree (hindx);
       mxfree (tempmat);
       mxfree (hargs);
       vfree (rho);
       vfree (rho_1);
       vfree (rho_2);
       return g;
}
```

A.2.15 Read_Parms.c

```
/* This function reads in the simulation parameters from a file
 * designated on the command line.
 *
 * The file should have one parameter per line, in this format:
 * <value> <variable>
 * For example, 512 G_b
 *
 * The order of the variables should be as follows:
 * ar_method(4), iidtype(2), Itype(2), limit(4), Bh, Eb_max,
 * Eb_min, I, iidparms[2], ISR, it, K, L, Nc, P, PG, sigmat,
 * T, z, freq[2], poly[2]
 */
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "mathlib.h"
#include "sim_parameters_extern.h"
intread_parms (char *filename)
{
   int i, k,                                                        /* Counter variables */
     L1,                                                            /* Lengths for recursive poly[] */
     numline,                                               /* Counter for current line being processed */
     space,                                                         /* Location of space in line */
     char_variables = 4,                    /* number of variables that are char arrays */
     int_variables = 16,                    /* number of variables that are ints */
     numcommands = 24,                      /* 26 variable assignments */
     maxlength = 20;                                /* Max length of a line */
   char commands[24][20];                   /* 22 variable assignments, each
                                                                                       * max length 19 chars + 1 for NULL */
   float j;                                                         /* temp storage */
   FILE *parms;                                             /* For opening parameter file */
   Real_Vector poly_temp, poly_temp2;
   Complex initval = {
          0, 0
   };                                                                       /* For vector initialization */
   if ((parms = fopen (filename, "r")) == NULL)  /* Open parameter file */
          {
              printf ("Cannot open parameter file.\n");
              exit (1);
       }
for (i = 0; i < numcommands; i++)     /* Read each line into array */
       fgets (commands[i], (maxlength - 1), parms);
/* Input character arrays first */
for (numline = 0; numline < char_variables; numline++)
       {
              space = 0;                                        /* Find SPACE character */
              for (i = 0; i < maxlength; i++)
                {
                       if (commands[numline][i] == ' ')
                              {
                                space = i;
                                break;
                              }
                }
              /* Assign string to defined character variables */
              switch (numline)
                {
                case 0:
                       for (i = 0; i < space; i++)
                              ar_method[i] = commands[numline][i];
                       break;
                case 1:
                       for (i = 0; i < space; i++)
                              iidtype[i] = commands[numline][i];
                       break;
                case 2:
                       for (i = 0; i < space; i++)
                              Itype[i] = commands[numline][i];
                       break;
                case 3:
                       for (i = 0; i < space; i++)
                              limit[i] = commands[numline][i];
                       break;
                }
       }
/* Read in Integer variables */
for (numline; (numline < (char_variables + int_variables)); numline++)
       {
              i = atoi (commands[numline]);   /* Convert char number to int */
              /* Assign int to correct variable */
              switch (numline)
                {
                case (4):
                       Bh = i;
                       break;
                case (5):
                       Eb_max = i;
                       break;
                case (6):
                       Eb_min = i;
                       break;
                case (7):
                       I = i;
                       break;
                case (8):
                       iidparms[0] = i;
                       break;
                case (9):
                       iidparms[1] = i;
                       break;
                case (10):
                       ISR = i;
                       break;
                case (11):
                       it = i;
                       break;
                case (12):
                       K = i;
                       break;
                case (13):
                       L = i;
                       break;
                case (14):
                       Nc = i;
                       break;
                case (15):
                       P = i;
                       break;
                case (16):
                       PG = i;
                       break;
                case (17):
                       sigmat = i;
                       break;
                case (18):
                       T = i;
                       break;
                case (19):
                        z = i;
                        break;
                      }
          }
   /* Read in float variables */
   for (numline; numline < numcommands; numline++)
          {
                 j = (float) atof (commands[numline]);   /* Convert char number to float */
                 /* Assign float to correct variable */
                 switch (numline)
                   {
                   case (20):
                          freq[0] = j;                    /* should be calculated, not input */
                          freq[0] = z * Bh * Nc;
                          break;
                   case (21):
                          freq[1] = j;                    /* should be calculated, not input */
                          freq[1] = z * Bh * Nc * 0.3;
                          break;
                   case (22):
                          polynomial[0] = j;
                          break;                                  /* If polynomial length > 2, change */
                   case (23):                             /* the recursive algorithm below */
                          polynomial[1] = j;
                          break;                                  /* as needed. */
                   }
          }
         /**************************************
         /* Finish Global Assignments */
   /* Spread spectrum parameters */
   Nb = PG * Nc;
   B = z * Bh;
   Ntot = B * Nb;
   /* AR Model Parameters */
   Bar = Bh / 64;
   Bg = Bh;
   /* Signal parameters & Thermal noise */
   sigmat2 = pow (sigmat, 2);
   Eb_sigma = valloc (NULL, (Eb_max - Eb_min + 1));
   for (i = 0; i < (Eb_max - Eb_min + 1); i++)
          {
                 j = (double) (Eb_min + i) / 10.0;
                 Eb_sigma[i] = (Real) pow (10.0, j);
          }
   /* Mixed jammer */
   ISRcw = ISRpb = ISR / 2;
   /* Create poly via recursively increasing the order of the filter to 2^L */
   L1 = 2;                                                        /* Set length of polynomial vector */
   poly_temp2 = valloc (NULL, (L1 + L1 - 1));
   poly_temp2[0] = polynomial[0];
   poly_temp2[1] = polynomial[1];
   for (i = 0; i < L; i++)
          {
                 poly_temp = valloc (NULL, (L1 + L1 - 1));
                 conv (poly_temp2, L1, poly_temp2, L1, poly_temp);       /* poly_temp = conv(.) */
                 vfree (poly_temp2);
                 poly_temp2 = valloc (NULL, L1 + L1 - 1);
                 vcopy (poly_temp2, poly_temp, L1 + L1 - 1);     /* Copy poly_temp to poly_temp2 */
                 vfree (poly_temp);
                 L1 = L1 + L1 - 1;                        /* Set to new length */
          }
   vfree (poly_temp2);
   poly = valloc (NULL, L1);       /* Create global vector */
   vcopy (poly, poly_temp2, L1);
   a_t = valloc (NULL, L1);
   vcopy (a_t, poly, L1);
   useinput_ = 1;
   vscale (a_t, L1, -1.0);
   useinput_ = 0;
   if (Itype[0] == 'p' && Itype[1] == 'b')       /* Is using PB interferer, P = number */
          P = L1 - 1;                                     /* of coefficients */
   /* End Global Assignments */
   /**************************************/
   fclose (parms);
   return 1;
}
```

A.2.16 Receiver.c

```
#include <math.h>
#include <stdlib.h>
#include "mathlib.h"
#include "sim_parameters_extern.h"
  void
receiver (Real_Vector signal, Real_Vector noise, char *limit,
                          Real_Vector rcvd_tr)
{
   int i;                                                         /* Counter */
   for (i = 0; i < Ntot; i++)
          *(rcvd_tr + i) = *(signal + i) + *(noise + i);
   if (limit[0] == 'h')            /* If limiter is a HARD LIMITER */
         {                          /* aka NOISE BLANKER */
                 for (i = 0; i < Ntot; i++)
                   {
                          if ((*(rcvd_tr + i) > T) || (*(rcvd_tr + i) < -T))
                                 *(rcvd_tr + i) = 0;
                   }
          }
   else                             /* If SOFT LIMITER */
          {
                 for (i = 0; i < Ntot; i++)
                   {
                          if ((*(rcvd_tr + i) > T))
                                 *(rcvd_tr + i) = T;
                          else if (*(rcvd_tr + i) < -T)
                                 *(rcvd_tr + i) = -T;
                   }
          }
}
```

A.2.17 Reshape.c

```
/***This is the C version of the Matlab Reshape() function****
***written by Tayo Ihimoyan, 01/2000******/
#include "cmt.h"
#include "mathlib.h"
/*---------------------------------------------------------------------------*/
Real_Matrix
reshape (Real_Matrix M, unsigned m1, unsigned n1, unsigned m2, unsigned n2)
{
/*
---------------------------------------------------------------------------
reshape() returns the m2-by-n2 matrix M2 whose elements
are taken column-wise from M(m1-by-n1).
   On exit, the output matrix M2[][] is the value of this routine.
         */
         /*---------------------------------------------------------------------------*/
   unsigned i, j, k, l;
   Real_Matrix M2;
   if (M == NULL)
          {
                 matherr_ ("reshape", E_NULLPTR);
                 return NULL;
          }
   /*returns an error message if m2*n2 is not equal to M elements */
   if (m1 * n1 != m2 * n2)
          {
                 matherr_ ("reshape", E_NULLPTR);
                 return NULL;
          }
   if ((M2 = mxalloc (NULL, m2, n2)) == NULL)
          return NULL;
   k = 0;
   l = 0;
   for (j = 0; j < n1; j++)
          {
                 for (i = 0; i < m1; i++)
                   {
                          if (l >= m2)
                                 {
                                   l = 0;
                                   k++;
                                 }
                          M2[l][k] = M[i][j];
                          l++;
                   }
          }
   return M2;
}
```

A.2.18 Th_Inter.c

```
/* interferer must be initialized to zero (or contain values from another
interference type) before using this function
*/
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include <time.h>
#include "mathlib.h"
#include "sim_parameters_extern.h"
voidth_inter (Real_Vector interferer)
{
   int k,                                                                 /* Counter */
     mu,                                                          /* Mean */
     sig;                                                                 /* Std dev */
   Complex pps,                                           /* Power per sample in the sequence */
     Iseq_conj;
   double white_mean, white_varp, white_vars, c;
   long ltime;                                            /* For seeding */
   Real_Vector white,                             /* White noise sequence */
     b;                                                                   /* numerator of filter */
   Complex_Vector Isequence;
   Real normalize, req;                   /* Required power level */
   if ((iidtype[0] == 'g') && (iidtype[1] == 's'))       /* Gaussian Case */
          {
                 mu = iidparms[0];                        /* Easier reading this way */
                 sig = iidparms[1];
                 white = valloc (NULL, Ntot);
                 ltime = time (NULL);
                 srand ((unsigned) ltime / 2);
                 for (k = 0; k < Ntot; k++)      /* Create white noise sequence */
                   *(white + k) = sig * normal (1, 0) - mu;
                 white_mean = stats (white, Ntot, &white_varp, &white_vars);     /* Get actual mean/var */
                 for (k = 0; k < Ntot; k++)      /* Make white EXACTLY Gaussian */
                   *(white + k) = (*(white + k) - white_mean) / sqrt (white_varp);
                 /* Build the interference signal using the AR model as a filter */
                 b = valloc (NULL, 1);
                 c = 192.6592;                            /* Maximum gain in poly filter (in dB) */
                 b[0] = pow (10.0, (-c / 20.0)); /* Filter with no zeros */
                 filter (b, 1, poly, P + 1, white, Ntot, interferer);
          }
   /* Find the total power per sample in the sequence, and normalize the interferer */
   Isequence = Cvalloc (NULL, Ntot);
   for (k = 0; k < Ntot; k++)      /* Copy real interferer to complex Isequence */
          {
                 Isequence[k].r = *(interferer + k);
                 Isequence[k].i = 0.0;
          }
   fft42 (Isequence, Ntot, -1);  /* Compute fft of Isequence */
   pps.i = 0.0;                                           /* Initialize power per sample to zero */
   pps.r = 0.0;
   for (k = 0; k < Ntot; k++)      /* Compute total power per sample */
          {
                 Iseq_conj = Conjg (*(Isequence + k));
                 pps = Cadd (pps, Cmul (Iseq_conj, *(Isequence + k)));
          }
   pps.r = (Real) pps.r / pow (Ntot, 2);
   req = pow (10, (ISR / 10));     /* Compute required power level */
   normalize = sqrt ((Real) req / pps.r);        /* Normalize interferer to ISR db */
   useinput_ = 1;                                         /* Perform normalization "in-place" */
   interferer = vscale (interferer, Ntot, normalize);
   useinput_ = 0;                                         /* Reset to original value */
   vfree (white);
   vfree (b);
   Cvfree (Isequence);
}
```

A.2.19 Toeplitz.c

```
/* Toeplitz
  * A Toeplitz matrix is defined by one column, col1, and one row, row1, of lengths numrows
  * and numcols, respectively.  If col1[0] != row1[0], then col1[0] will override.
  */
#include "mathlib.h"
#include <stdlib.h>
Real_Matrix
toeplitz (Real_Vector col1, Real_Vector row1, unsigned int numrows,
                          unsigned int numcols)
{
   register int column, row;       /* counter variables */
   Real_Matrix toepmat;                   /* Output matrix, in Toeplitz form */
   toepmat = mxalloc (NULL, numrows, numcols);   /* Create matrix */
   for (column = 0; column < numcols; column++)  /* Fill in Toeplitz form into matrix */
          {
                 if (column == 0)                         /* Set first column */
                   {
                          for (row = 0; row < numrows; row++)
                                 toepmat[row][column] = *(col1 + row);   /* Set first column */
                   }
                 else
                   {
                          toepmat[0][column] = *(row1 + column); /* Set first row */
                          for (row = 1; row < numrows; row++)    /* Use shift register to do rest of rows */
                                 toepmat[row][column] = toepmat[row - 1][column - 1];
                   }
          }
   return toepmat;
}
```

Appendix B

MATLAB® Plotting Script

```
% C Simulation Plotting Routine
%
% Spetember 16, 1999
% Michael Banys
%c_string_creation;
% Load strings for use in plots
do_spectrum = 1;    % Does spectral plot if do_spectrum == 1
% OPEN FILES
pbar = fopen('pbar.txt','r');
pblc = fopen('pblc.txt','r');
pblo = fopen('pblo.txt','r');
pblw = fopen('pblw.txt','r');
pbwc = fopen('pbwc.txt','r');
snr_vals = fopen('snr_vals_.txt','r');
% READ DATA
snr_list = fscanf(snr_vals,'%f',inf);
Pb_ar_avg = fscanf(pbar,'%f',inf);
Pb_lc_avg = fscanf(pblc,'%f',inf);
Pb_lo_avg = fscanf(pblo,'%f',inf);
Pb_lw_avg = fscanf(pblw,'%f',inf);
Pb_wc_avg = fscanf(pbwc,'%f',inf);
%%%%%%%%%%%%% PBPLOT %%%%%%%%%%%%%%%%%
figure;
hold on;
% May need to use plot & log10(.)
title(strcat(plot_title1,INTERFERER,plot_title2));
xlabel('Signal-to-Thermal Noise Ratio (dB)');
ylabel('Probability of Bit Error (10^x)');
plot(snr_list, log10(Pb_ar_avg), 'x-');
plot(snr_list, log10(Pb_wc_avg), ':');
plot(snr_list, log10(Pb_lo_avg), 'o');
plot(snr_list, log10(Pb_lw_avg), '--');
plot(snr_list, log10(Pb_lc_avg), '-');
legend(gca, 'ARLO', 'Correlator on whitened', 'LO detector', 'LO on whitened', 'Linear Correlator');
set(gca,'YTick',[-7 -6 -5 -4 -3 -2 -1 0 1]);
message = sprintf('Make note of all the important variables (ISR, Bh, P, etc.)');
disp(message);
hold off;
%.........................................................
if do_spectrum == 1
%%%%%%%%%%%%%% SPPLOT %%%%%%%%%%%%%%%%%
interferer_out = fopen('interferer_out.txt','r');
noise_out = fopen('noise_out.txt','r');
rcvd_tr_out = fopen('rcvd_tr_out.txt','r');
a_hat_out = fopen('a_hat_out.txt','r');
interferer = fscanf(interferer_out,'%f',inf);
noise = fscanf(noise_out,'%f',inf);
rcvd_tr = fscanf(rcvd_tr_out,'%f',inf);
a_hat =fscanf(a_hat_out,'%f',inf);
Ptt = spectrum(noise - interferer);     % Spectrum of thermal noise
Ptt = 10*log10(Ptt(:,1));               % Remove phase info and scale
delta_f = 1/(length(Ptt) - 1);          % For appropriately scaling the frequency axis
f = 0:(length(Ptt) - 1);                % Obtain frequency components
f = f * delta_f;                        % Scale accordingly
Pss = spectrum(noise - rcvd_tr);        % Spectrum of transmitted signal
Pss = 10*log10(Pss(:,1));               % Remove phase info and scale
Pii = spectrum(interferer);             % Spectrum of interference signal
Pii = 10*log10(Pii(:,1));               % Remove phase info and scale
Prr = spectrum(rcvd_tr);                % Spectrum of received signal
Prr = 10*log10(Prr(:,1));               % Remove phase info and scale
Pww = spectrum(filter(-a_hat, 1, rcvd_tr));     % Spectrum of whitened signal
Pww = 10*log10(Pww(:,1));                 % Remove phase info and scale
figure;
hold on;
title(strcat(plot_title3,INTERFERER,plot_title2));
xlabel('Frequency (with respect to sampling frequency)');
ylabel('Power Magnitude (dB)');
```

Bibliography

[1] W. E. Jacklin, "Statistical Methods for Robust Locally Optimum Signal Detection," Ph. D. Dissertation, The Illinois Institute of Technology, Jul. 1996.

[2] D. R. Ucci, W. E. Jacklin, J. H. Grimm, *A Spread Spectrum Receiver With Nonlinear Processing*, Final Technical Report for Rome Laboratory, USAF Report No. RL-TR-93-50, 1993.

[3] M. R. Mychal and D. R. Ucci, "Simulation of a Robust Locally Optimum Receiver in Correlated Noise Using Autoregressive Modeling," AFOSR Summer Research Program Final Report, September, 1997.

[4] K. S. Shanmugan, A. M. Breipohl, *Random Signals.* John Wiley & Sons. 1988. Page 484.

[5] L. Sardo, J Kittler, *Minimum Complexity PDF Estimation for Correlated Data*, Proceedings of the 13th International Conference on Pattern Recognition, 1996, Volume 2, Pages 750-754.

[6] Matlab Documentation: Matlab Compiler User's Guide *Version 2.0.* January 1999. The MathWorks, Inc.

[7] M. R. Mychal, W. E. Jacklin, D. R. Ucci, A. L. Kaczmarski, and M. R. Banys, "Robust Locally Optimum Signal Detection in Correlated Noise Using Autoregressive Modeling," Proc. Military Communications International Symposium '99, October, 1999.